Ice Manual

Distributed Programming with Ice

The Internet Communications Engine (Ice) is an object-oriented RPC framework that helps you build distributed applications with minimal effort. Ice allows you to focus your efforts on your application logic, and it takes care of all interactions with low-level network programming interfaces. With Ice, there is no need to worry about details such as opening network connections, serializing and deserializing data for network transmission, or retrying failed connection attempts.

The main design goals of Ice are:

- Provide an object-oriented RPC framework suitable for use in heterogeneous environments.
- Provide a full set of features that support development of realistic distributed applications for a wide variety of domains.
- Avoid unnecessary complexity, making the platform easy to learn and to use.
- Provide an implementation that is efficient in network bandwidth, memory use, and CPU overhead.
- Provide an implementation that has built-in security, making it suitable for use over insecure public networks.

In simpler terms, the Ice design goals could be stated as "Let's build a powerful middleware platform that makes the developer's life easier."

> The acronym "Ice" is pronounced as a single syllable, like the word for frozen water.

Getting Help with Ice

If you have a question and you cannot find an answer in this manual, you can visit our developer forums to see if another developer has encountered the same issue. If you still need help, feel free to post your question on the forum, which ZeroC's developers monitor regularly. Note, however, that we can provide only limited free support in our forums. For guaranteed response and problem resolution times, we highly recommend purchasing commercial support.

Feedback about the Manual

We would very much like to hear from you in case you find any bugs (however minor) in this manual. We also would like to hear your opinion on the contents, and any suggestions as to how it might be improved. You can contact us via e-mail at icebook@zeroc.com.

Legal Notices

License

Copyright

mailto:info@zeroc.com
https://zeroc.com

Ice Overview

The following topics provide a high-level overview of Ice:

- Ice Architecture introduces fundamental concepts and terminology, and outlines how Slice definitions, language mappings, and the Ice run time and protocol work in concert to create clients and servers.
- Ice Services Overview briefly presents the object services provided by Ice.
- Architectural Benefits of Ice outlines the benefits that result from the Ice architecture.

Topics

Ice Architecture

Ice is an object-oriented middleware platform. Fundamentally, this means that Ice provides tools, APIs, and library support for building object-oriented client-server applications. Ice applications are suitable for use in heterogeneous environments: client and server can be written in different programming languages, can run on different operating systems and machine architectures, and can communicate using a variety of networking technologies. The source code for these applications is portable regardless of the deployment environment.

Topics:

See Also

Terminology

Every computing technology creates its own vocabulary as it evolves. Ice is no exception. However, the amount of new jargon used by Ice is minimal. Rather than inventing new terms, we have used existing terminology as much as possible. If you have used another middleware technology in the past, you will be familiar with much of what follows. (However, we suggest you at least skim the material because a few terms used by Ice *do* differ from the corresponding terms used by other middleware.)

On this page:

Clients and Servers

The terms *client* and *server* are not firm designations for particular parts of an application; rather, they denote roles that are taken by parts of an application for the duration of a request:

- Clients are active entities. They issue requests for service to servers.
- Servers are passive entities. They provide services in response to client requests.

Frequently, servers are not "pure" servers, in the sense that they never issue requests and only respond to requests. Instead, servers often act as a server on behalf of some client but, in turn, act as a client to another server in order to satisfy their client's request.

Similarly, clients often are not "pure" clients, in the sense that they only request service from an object. Instead, clients are frequently client-server hybrids. For example, a client might start a long-running operation on a server; as part of starting the operation, the client can provide a *callback object* to the server that is used by the server to notify the client when the operation is complete. In that case, the client acts as a client when it starts the operation, and as a server when it is notified that the operation is complete.

Such role reversal is common in many systems, so, frequently, client-server systems could be more accurately described as *peer-to-peer* sy stems.

Ice Objects

An *Ice object* is a conceptual entity, or abstraction. An Ice object can be characterized by the following points:

- An Ice object is an entity in the local or a remote address space that can respond to client requests.
- A single Ice object can be instantiated in a single server or, redundantly, in multiple servers. If an object has multiple simultaneous instantiations, it is still a single Ice object.
- Each Ice object has one or more *interfaces*. An interface is a collection of named *operations* that are supported by an object. Clients issue requests by invoking operations.
- An operation has zero or more *parameters* as well as a *return value*. Parameters and return values have a specific *type*. Parameters are named and have a direction: in-parameters are initialized by the client and passed to the server; out-parameters are initialized by the server and passed to the client. (The return value is simply a special out-parameter.)
- An Ice object has a distinguished interface, known as its *main interface*. In addition, an Ice object can provide zero or more alternate interfaces, known as *facets*. Clients can select among the facets of an object to choose the interface they want to work with.

- Each Ice object has a unique *object identity*. An object's identity is an identifying value that distinguishes the object from all other objects. The Ice object model assumes that object identities are globally unique, that is, no two objects within an Ice communication domain can have the same object identity.

  In practice, you need not use object identities that are globally unique, such as UUIDs, only identities that do not clash with any other identity within your domain of interest. However, there are architectural advantages to using globally unique identifiers, which we explore in our discussion of object life cycle.

Proxies

For a client to be able to contact an Ice object, the client must hold a *proxy* for the Ice object. A proxy is an artifact that is local to the client's address space; it represents the (possibly remote) Ice object for the client. A proxy acts as the local ambassador for an Ice object: when the client invokes an operation on the proxy, the Ice run time:

1. Locates the Ice object
2. Activates the Ice object's server if it is not running
3. Activates the Ice object within the server
4. Transmits any in-parameters to the Ice object
5. Waits for the operation to complete
6. Returns any out-parameters and the return value to the client (or throws an exception in case of an error)

A proxy encapsulates all the necessary information for this sequence of steps to take place. In particular, a proxy contains:

- Addressing information that allows the client-side run time to contact the correct server
- An object identity that identifies which particular object in the server is the target of a request
- An optional facet identifier that determines which particular facet of an object the proxy refers to

Stringified Proxies

The information in a proxy can be expressed as a string. For example, the string:

```
SimplePrinter:default -p 10000
```

is a human-readable representation of a proxy. The Ice run time provides API calls that allow you to convert a proxy to its stringified form and vice versa. This is useful, for example, to store proxies in database tables or text files.

Provided that a client knows the identity of an Ice object and its addressing information, it can create a proxy "out of thin air" by supplying that information. In other words, no part of the information inside a proxy is considered opaque; a client needs to know only an object's identity, addressing information, and (to be able to invoke an operation) the object's type in order to contact the object.

Direct Proxies

A *direct proxy* is a proxy that embeds an object's identity, together with the address at which its server runs. The address is completely specified by:

- a protocol identifier (such TCP/IP or UDP)
- a protocol-specific address (such as a host name and port number)

To contact the object denoted by a direct proxy, the Ice run time uses the addressing information in the proxy to contact the server; the identity of the object is sent to the server with each request made by the client.

Indirect Proxies

An *indirect proxy* has two forms. It may provide only an object's identity, or it may specify an identity together with an object adapter identifier. An object that is accessible using only its identity is called a well-known object, and the corresponding proxy is a well-known proxy. For example, the string:

```
SimplePrinter
```

is a valid proxy for a well-known object with the identity `SimplePrinter`.

An indirect proxy that includes an object adapter identifier has the stringified form

```
SimplePrinter@PrinterAdapter
```

Any object of the object adapter can be accessed using such a proxy, regardless of whether that object is also a well-known object.

Notice that an indirect proxy contains no addressing information. To determine the correct server, the client-side run time passes the proxy information to a location service. In turn, the location service uses the object identity or the object adapter identifier as the key in a lookup table that contains the address of the server and returns the current server address to the client. The client-side run time now knows how to contact the server and dispatches the client request as usual.

The entire process is similar to the mapping from Internet domain names to IP address by the Domain Name Service (DNS): when we use a domain name, such as `www.zeroc.com`, to look up a web page, the host name is first resolved to an IP address behind the scenes and, once the correct IP address is known, the IP address is used to connect to the server. With Ice, the mapping is from an object identity or object adapter identifier to a protocol-address pair, but otherwise very similar. The client-side run time knows how to contact the location service via configuration (just as web browsers know which DNS server to use via configuration).

Direct Versus Indirect Binding

The process of resolving the information in a proxy to protocol-address pair is known as *binding*. Not surprisingly, *direct binding* is used for direct proxies, and *indirect binding* is used for indirect proxies.

The main advantage of indirect binding is that it allows us to move servers around (that is, change their address) without invalidating existing proxies that are held by clients. In other words, direct proxies avoid the extra lookup to locate the server but no longer work if a server is moved to a different machine. On the other hand, indirect proxies continue to work even if we move (or *migrate*) a server.

Fixed Proxies

A *fixed proxy* is a proxy that is bound to a particular connection: instead of containing addressing information or an adapter name, the proxy contains a connection handle. The connection handle stays valid only for as long as the connection stays open so, once the connection is closed, the proxy no longer works (and will never work again). Fixed proxies cannot be marshaled, that is, they cannot be passed as parameters on operation invocations. Fixed proxies are used to allow bidirectional communication, so a server can make callbacks to a client without having to open a new connection.

Routed Proxies

A *routed proxy* is a proxy that forwards all invocations to a specific target object, instead of sending invocations directly to the actual target. Routed proxies are useful for implementing services such as Glacier2, which enables clients to communicate with servers that are behind a firewall.

Replication

In Ice, *replication* involves making object adapters (and their objects) available at multiple addresses. The goal of replication is usually to provide redundancy by running the same server on several computers. If one of the computers should happen to fail, a server still remains available on the others.

The use of replication implies that applications are designed for it. In particular, it means a client can access an object via one address and obtain the same result as from any other address. Either these objects are stateless, or their implementations are designed to synchronize with a database (or each other) in order to maintain a consistent view of each object's state.

Ice supports a limited form of replication when a proxy specifies multiple addresses for an object. The Ice run time selects one of the addresses at random for its initial connection attempt and tries all of them in the case of a failure. For example, consider this proxy:

```
SimplePrinter:tcp -h server1 -p 10001:tcp -h server2 -p 10002
```

The proxy states that the object with identity `SimplePrinter` is available using TCP at two addresses, one on the host `server1` and another on the host `server2`. The burden falls to users or system administrators to ensure that the servers are actually running on these computers at the specified ports.

Replica Groups

In addition to the proxy-based replication described above, Ice supports a more useful form of replication known as *replica groups* that requires the use of a location service.

A replica group has a unique identifier and consists of any number of object adapters. An object adapter may be a member of at most one replica group; such an adapter is considered to be a *replicated object adapter*.

After a replica group has been established, its identifier can be used in an indirect proxy in place of an adapter identifier. For example, a replica group identified as `PrinterAdapters` can be used in a proxy as shown below:

```
SimplePrinter@PrinterAdapters
```

The replica group is treated by the location service as a "virtual object adapter." The behavior of the location service when resolving an indirect proxy containing a replica group id is an implementation detail. For example, the location service could decide to return the addresses of all object adapters in the group, in which case the client's Ice run time would select one of the addresses at random using the limited form of replication discussed earlier. Another possibility is for the location service to return only one address, which it decided upon using some heuristic.

Regardless of the way in which a location service resolves a replica group, the key benefit is indirection: the location service as a middleman can add more intelligence to the binding process.

Servants

As we mentioned, an Ice Object is a conceptual entity that has a type, identity, and addressing information. However, client requests ultimately must end up with a concrete server-side processing entity that can provide the behavior for an operation invocation. To put this differently, a client request must ultimately end up executing code inside the server, with that code written in a specific programming language and executing on a specific processor.

The server-side artifact that provides behavior for operation invocations is known as a *servant*. A servant provides substance for (or *incarnates*) one or more Ice objects. In practice, a servant is simply an instance of a class that is written by the server developer and that is registered with the server-side run time as the servant for one or more Ice objects. Methods on the class correspond to the operations on the Ice object's interface and provide the behavior for the operations.

A single servant can incarnate a single Ice object at a time or several Ice objects simultaneously. If the former, the identity of the Ice object incarnated by the servant is implicit in the servant. If the latter, the servant is provided the identity of the Ice object with each request, so it can decide which object to incarnate for the duration of the request.

Conversely, a single Ice object can have multiple servants. For example, we might choose to create a proxy for an Ice object with two different addresses for different machines. In that case, we will have two servers, with each server containing a servant for the same Ice object. When a client invokes an operation on such an Ice object, the client-side run time sends the request to exactly one server. In other words, multiple servants for a single Ice object allow you to build redundant systems: the client-side run time attempts to send the request to one server and, if that attempt fails, sends the request to the second server. An error is reported back to the client-side application code only if that second attempt also fails.

At-Most-Once Semantics

Ice requests have *at-most-once* semantics: the Ice run time does its best to deliver a request to the correct destination and, depending on the exact circumstances, may retry a failed request. Ice guarantees that it will either deliver the request, or, if it cannot deliver the request, inform the client with an appropriate exception; under no circumstances is a request delivered twice, that is, retries are attempted only if it is known that a previous attempt definitely failed.

> One exception to this rule are datagram invocations over UDP transports. For these, duplicated UDP packets can lead to a violation of at-most-once semantics.

At-most-once semantics are important because they guarantee that operations that are not *idempotent* can be used safely. An idempotent

operation is an operation that, if executed twice, has the same effect as if executed once. For example, `x = 1;` is an idempotent operation: if we execute the operation twice, the end result is the same as if we had executed it once. On the other hand, `x++;` is not idempotent: if we execute the operation twice, the end result is not the same as if we had executed it once.

Without at-most-once semantics, we can build distributed systems that are more robust in the presence of network failures. However, realistic systems require non-idempotent operations, so at-most-once semantics are a necessity, even though they make the system less robust in the presence of network failures. Ice permits you to mark individual operations as idempotent. For such operations, the Ice run time uses a more aggressive error recovery mechanism than for non-idempotent operations.

Synchronous Method Invocation

By default, the request dispatch model used by Ice is a synchronous remote procedure call: an operation invocation behaves like a local procedure call, that is, the client thread is suspended for the duration of the call and resumes when the call completes (and all its results are available).

Asynchronous Method Invocation

Ice also supports *asynchronous method invocation* (AMI): a client can invoke operations *asynchronously*, which means the client's calling thread does not block while waiting for the invocation to complete. The client passes the normal parameters and, depending on the language mapping, might also pass a callback that the client-side run time invokes upon completion, or the invocation might return a future that the client can eventually use to obtain the results.

The server cannot distinguish an asynchronous invocation from a synchronous one — either way, the server simply sees that a client has invoked an operation on an object.

Asynchronous Method Dispatch

Asynchronous method dispatch (AMD) is the server-side equivalent of AMI. For synchronous dispatch (the default), the server-side run time up-calls into the application code in the server in response to an operation invocation. While the operation is executing (or sleeping, for example, because it is waiting for data), a thread of execution is tied up in the server; that thread is released only when the operation completes.

With asynchronous method dispatch, the server-side application code is informed of the arrival of an operation invocation. However, instead of being forced to process the request immediately, the server-side application can choose to delay processing of the request and, in doing so, releases the execution thread for the request. The server-side application code is now free to do whatever it likes. Eventually, once the results of the operation are available, the server-side application code makes an API call to inform the server-side Ice run time that a request that was dispatched previously is now complete; at that point, the results of the operation are returned to the client.

Asynchronous method dispatch is useful if, for example, a server offers operations that block clients for an extended period of time. For example, the server may have an object with a `get` operation that returns data from an external, asynchronous data source and that blocks clients until the data becomes available. With synchronous dispatch, each client waiting for data to arrive ties up an execution thread in the server. Clearly, this approach does not scale beyond a few dozen clients. With asynchronous dispatch, hundreds or thousands of clients can be blocked in the same operation invocation without tying up any threads in the server.

Synchronous and asynchronous method dispatch are transparent to the client, that is, the client cannot tell whether a server chose to process a request synchronously or asynchronously.

Oneway Method Invocation

Clients can invoke an operation as a *oneway* operation. A oneway invocation has "best effort" semantics. For a oneway invocation, the client-side run time hands the invocation to the local transport, and the invocation completes on the client side as soon as the local transport has buffered the invocation. The actual invocation is then sent asynchronously by the operating system. The server does not reply to oneway invocations, that is, traffic flows only from client to server, but not vice versa.

Oneway invocations are unreliable. For example, the target object may not exist, in which case the invocation is simply lost. Similarly, the operation may be dispatched to a servant in the server, but the operation may fail (for example, because parameter values are invalid); if so, the client receives no notification that something has gone wrong.

Oneway invocations are possible only on operations that do not have a return value, do not have out-parameters, and do not throw user exceptions.

To the application code on the server-side, oneway invocations are transparent, that is, there is no way to distinguish a twoway invocation from a oneway invocation.

Oneway invocations are available only if the target object offers a stream-oriented transport, such as TCP/IP or SSL.

Note that, even though oneway operations are sent over a stream-oriented transport, they may be processed out of order in the server. This can happen because each invocation may be dispatched in its own thread: even though the invocations are *initiated* in the order in which the invocations arrive at the server, this does not mean that they will be *processed* in that order — the vagaries of thread scheduling can result in a oneway invocation completing before other oneway invocations that were received earlier.

Batched Oneway Method Invocation

Each oneway invocation sends a separate message to the server. For a series of short messages, the overhead of doing so is considerable: the client- and server-side run time each must switch between user mode and kernel mode for each message and, at the networking level, each message incurs the overheads of flow-control and acknowledgement.

Batched oneway invocations allow you to send a series of oneway invocations as a single message: every time you invoke a batched oneway operation, the invocation is buffered in the client-side run time. Once you have accumulated all the oneway invocations you want to send, you make a separate API call to send all the invocations at once. The client-side run time then sends all of the buffered invocations in a single message, and the server receives all of the invocations in a single message. This avoids the overhead of repeatedly trapping into the kernel for both client and server, and is much easier on the network between them because one large message can be transmitted more efficiently than many small ones.

The individual invocations in a batched oneway message are dispatched by a single thread in the order in which they were placed into the batch. This guarantees that the individual operations in a batched oneway message are processed in order in the server.

Batched oneway invocations are particularly useful for messaging services, such as IceStorm, and for fine-grained interfaces that offer `set` operations for small attributes.

Datagram Invocations

Datagram invocations have "best effort" semantics similar to oneway invocations. However, datagram invocations require the object to offer UDP as a transport (whereas oneway invocations require TCP/IP).

Like a oneway invocation, a datagram invocation can be made only if the operation does not have a return value, out-parameters, or user exceptions. A datagram invocation uses UDP to invoke the operation. The operation returns as soon as the local UDP stack has accepted the message; the actual operation invocation is sent asynchronously by the network stack behind the scenes.

Datagrams, like oneway invocations, are unreliable: the target object may not exist in the server, the server may not be running, or the operation may be invoked in the server but fail due to invalid parameters sent by the client. As for oneway invocations, the client receives no notification of such errors.

However, unlike oneway invocations, datagram invocations have a number of additional error scenarios:

- Individual invocations may simply be lost in the network.
  This is due to the unreliable delivery of UDP packets. For example, if you invoke three operations in sequence, the middle invocation may be lost. (The same thing cannot happen for oneway invocations — because they are delivered over a connection-oriented transport, individual invocations cannot be lost.)
- Individual invocations may arrive out of order.
  Again, this is due to the nature of UDP datagrams. Because each invocation is sent as a separate datagram, and individual datagrams can take different paths through the network, it can happen that invocations arrive in an order that differs from the order in which they were sent.

Datagram invocations are well suited for small messages on LANs, where the likelihood of loss is small. They are also suited to situations in which low latency is more important than reliability, such as for fast, interactive internet applications. Finally, datagram invocations can be used to multicast messages to multiple servers simultaneously.

Batched Datagram Invocations

As for batched oneway invocations, *batched datagram invocations* allow you to accumulate a number of invocations in a buffer and then send the entire buffer as a single datagram by making an API call to flush the buffer. Batched datagrams reduce the overhead of repeated system calls and allow the underlying network to operate more efficiently. However, batched datagram invocations are useful only for batched messages whose total size does not substantially exceed the PDU limit of the network: if the size of a batched datagram gets too large, UDP fragmentation makes it more likely that one or more fragments are lost, which results in the loss of the entire batched message. However, you are guaranteed that either all invocations in a batch will be delivered, or none will be delivered. It is impossible for individual invocations within a batch to be lost.

Batched datagrams use a single thread in the server to dispatch the individual invocations in a batch. This guarantees that the invocations

are made in the order in which they were queued — invocations cannot appear to be reordered in the server.

Run-Time Exceptions

Any operation invocation can raise a *run-time exception*. Run-time exceptions are pre-defined by the Ice run time and cover common error conditions, such as connection failure, connection timeout, or resource allocation failure. Run-time exceptions are presented to the application as native exceptions and so integrate neatly with the native exception handling capabilities of languages that support exception handling.

User Exceptions

A server indicates application-specific error conditions by raising *user exceptions* to clients. User exceptions can carry an arbitrary amount of complex data and can be arranged into inheritance hierarchies, which makes it easy for clients to handle categories of errors generically, by catching an exception that is further up the inheritance hierarchy. Like run-time exceptions, user exceptions map to native exceptions.

Properties

Much of the Ice run time is configurable via *properties*. Properties are name-value pairs, such as `Ice.Default.Protocol=tcp`. Properties are typically stored in text files and parsed by the Ice run time to configure various options, such as the thread pool size, the level of tracing, and various other configuration parameters.

See Also

- The Slice Language
- Proxies for Ice Objects
- Locators
- Object Life Cycle
- Bidirectional Connections
- Glacier2
- IceStorm

Slice (Specification Language for Ice)

Each Ice object has an interface with a number of operations. Interfaces, operations, and the types of data that are exchanged between client and server are defined using the *Slice language*. Slice allows you to define the client-server contract in a way that is independent of a specific programming language, such as C++, Java, or C#. The Slice definitions are compiled by a compiler into an API for a specific programming language, that is, the part of the API that is specific to the interfaces and types you have defined consists of generated code.

See Also

- The Slice Language

Overview of the Language Mappings

The rules that govern how each Slice construct is translated into a specific programming language are known as *language mappings*. For example, for the C++ mapping, a Slice sequence appears as a `std::vector`, whereas, for the Java mapping, a Slice sequence appears as a Java array. In order to determine what the API for a specific Slice construct looks like, you only need the Slice definition and knowledge of the language mapping rules. The rules are simple and regular enough to make it unnecessary to read the generated code to work out how to use the generated API.

Of course, you are free to peruse the generated code. However, as a rule, that is inefficient because the generated code is not necessarily suitable for human consumption. We recommend that you familiarize yourself with the language mapping rules; that way, you can mostly ignore the generated code and need to refer to it only when you are interested in some specific detail.

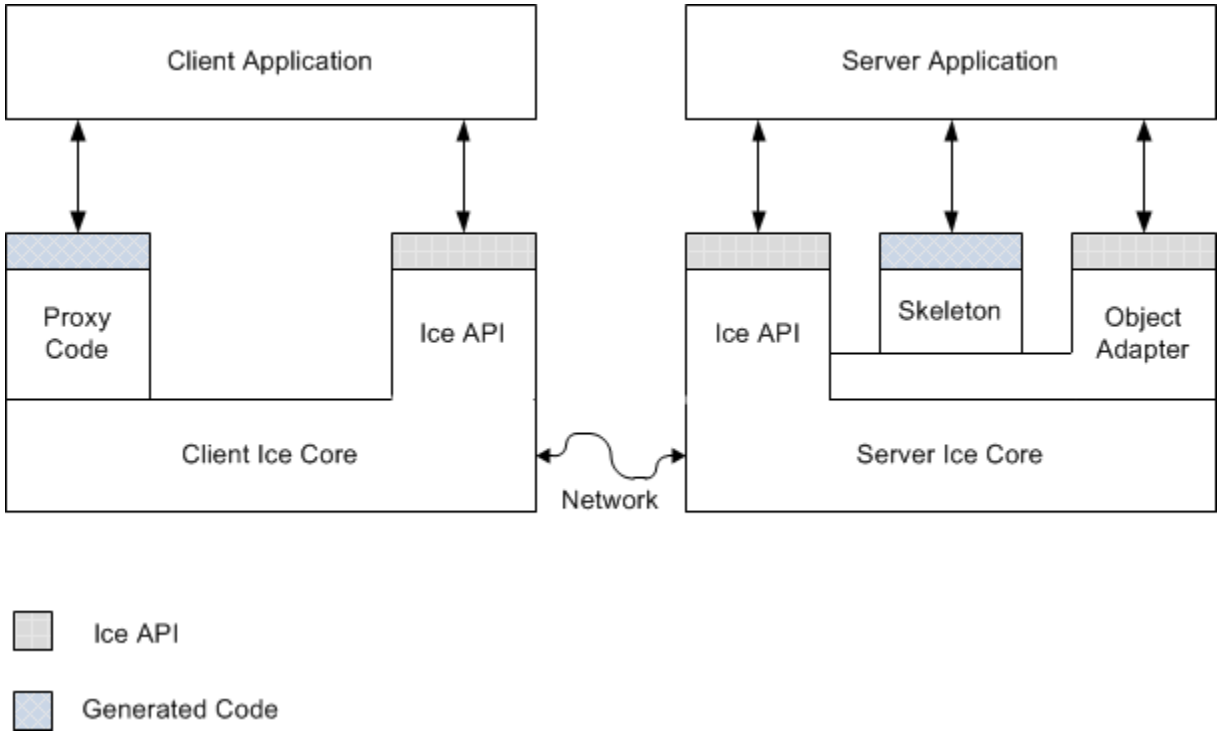
Currently, Ice provides language mappings for C++, C#, Java, JavaScript, Python, Objective-C, and, for the client side, PHP and Ruby.

See Also

- C++ Mapping
- C# Mapping
- Java Mapping
- JavaScript Mapping
- Objective-C Mapping
- PHP Mapping
- Python Mapping
- Ruby Mapping

Client and Server Structure

Ice clients and servers have the logical internal structure:



Ice Client and Server Structure

Both client and server consist of a mixture of application code, library code, and code generated from Slice definitions:

- The Ice core contains the client- and server-side run-time support for remote communication. Much of this code is concerned with the details of networking, threading, byte ordering, and many other networking-related issues that we want to keep away from application code. The Ice core is provided as a number of libraries that client and server use.
- The generic part of the Ice core (that is, the part that is independent of the specific types you have defined in Slice) is accessed through the Ice API. You use the Ice API to take care of administrative chores, such as initializing and finalizing the Ice run time. The Ice API is identical for clients and servers (although servers use a larger part of the API than clients).
- The proxy code is generated from your Slice definitions and, therefore, specific to the types of objects and data you have defined in Slice. The proxy code has two major functions:
  - It provides a down-call interface for the client. Calling a function in the generated proxy API ultimately ends up sending an RPC message to the server that invokes a corresponding function on the target object.
  - It provides *marshaling* and *unmarshaling* code. Marshaling is the process of serializing a complex data structure, such as a sequence or a dictionary, for transmission on the wire. The marshaling code converts data into a form that is standardized for transmission and independent of the endian-ness and padding rules of the local machine. Unmarshaling is the reverse of marshaling, that is, deserializing data that arrives over the network and reconstructing a local representation of the data in types that are appropriate for the programming language in use.
- The skeleton code is also generated from your Slice definition and, therefore, specific to the types of objects and data you have defined in Slice. The skeleton code is the server-side equivalent of the client-side proxy code: it provides an up-call interface that permits the Ice run time to transfer the thread of control to the application code you write. The skeleton also contains marshaling and unmarshaling code, so the server can receive parameters sent by the client, and return parameters and exceptions to the client.
- The object adapter is a part of the Ice API that is specific to the server side: only servers use object adapters. An object adapter has several functions:
  - The object adapter maps incoming requests from clients to specific methods on programming-language objects. In other words, the object adapter tracks which servants with what object identity are in memory.
  - The object adapter is associated with one or more transport endpoints. If more than one transport endpoint is associated with an adapter, the servants incarnating objects within the adapter can be reached via multiple transports. For example, you can associate both a TCP/IP and a UDP endpoint with an adapter, to provide alternate quality-of-service and performance characteristics.
  - The object adapter is responsible for the creation of proxies that can be passed to clients. The object adapter knows about

the type, identity, and transport details of each of its objects and embeds the correct details when the server-side application code requests the creation of a proxy.

Note that, as far as the process view is concerned, there are only two processes involved: the client and the server. All the run time support for distributed communication is provided by the Ice libraries and the code that is generated from Slice definitions. (For indirect proxies, a location service is required to resolve proxies to transport endpoints.)

See Also

- Hello World Application
- Locators

Overview of the Ice Protocol

Ice provides an RPC protocol that can use a variety of underlying transports. The most common examples are TCP and UDP, but Ice also supports Websocket, Bluetooth, and Apple's iAP. In addition, Ice allows you to use SSL as a transport, so all communication between client and server is encrypted.

The Ice protocol defines:

- a number of message types, such as request and reply message types,
- a protocol state machine that determines in what sequence different message types are exchanged by client and server, together with the associated connection establishment and tear-down semantics for TCP/IP,
- encoding rules that determine how each type of data is represented on the wire,
- a header for each message type that contains details such as the message type, the message size, and the protocol and encoding version in use.

Ice also supports compression on the wire: by setting a configuration parameter, you can arrange for all network traffic to be compressed to conserve bandwidth. This is useful if your application exchanges large amounts of data between client and server.

The Ice protocol is suitable for building highly-efficient event forwarding mechanisms because it permits forwarding of a message without knowledge of the details of the information inside a message. This means that messaging switches need not do any unmarshaling and remarshaling of messages — they can forward a message by simply treating it as an opaque buffer of bytes.

The Ice protocol also supports bidirectional operation: if a server wants to send a message to a callback object provided by the client, the callback can be made over the connection that was originally created by the client. This feature is especially important when the client is behind a firewall that permits outgoing connections, but not incoming connections.

See Also

- The Ice Protocol
- IceSSL
- Bidirectional Connections

Ice Services Overview

The Ice core provides a sophisticated client-server platform for distributed application development. However, realistic applications usually require more than just a remoting capability: typically, you also need a way to start servers on demand, distribute proxies to clients, distribute asynchronous events, configure your application, distribute patches for an application, and so on.

Ice ships with a number of services that provide these and other features. The services are implemented as Ice servers to which your application acts as a client. None of the services use Ice-internal features that are hidden from application developers so, in theory, you could develop equivalent services yourself. However, having these services available as part of the platform allows you to focus on application development instead of having to build a lot of infrastructure first. Moreover, building such services is not a trivial effort, so it pays to know what is available and use it instead of reinventing your own wheel.

On this page:

IceGrid

IceGrid is an implementation of an Ice location service that resolves the symbolic information in an indirect proxy to a protocol-address pair for indirect binding. A location service is only the beginning of IceGrid's capabilities.

IceGrid:

- allows you to register servers for automatic start-up: instead of requiring a server to be running at the time a client issues a request, IceGrid starts servers on demand, when the first client request arrives.
- provides tools that make it easy to configure complex applications containing several servers.
- supports replication and load-balancing.
- automates the distribution and patching of server executables and dependent files.
- provides a simple query service that allows clients to obtain proxies for objects they are interested in.

IceStorm

IceStorm is a publish-subscribe service that decouples clients and servers. Fundamentally, IceStorm acts as a distribution switch for events. Publishers send events to the service, which, in turn, passes the events to subscribers. In this way, a single event published by a publisher can be sent to multiple subscribers. Events are categorized by topic, and subscribers specify the topics they are interested in. Only events that match a subscriber's topic are sent to that subscriber. The service permits selection of a number of quality-of-service criteria to allow applications to choose the appropriate trade-off between reliability and performance.

IceStorm is particularly useful if you have a need to distribute information to large numbers of application components. (A typical example is a stock ticker application with a large number of subscribers.) IceStorm decouples the publishers of information from subscribers and takes care of the redistribution of the published events. In addition, IceStorm can be run as a *federated* service, that is, multiple instances of the service can be run on different machines to spread the processing load over a number of CPUs.

IcePatch2

IcePatch2 is a software patching service. It allows you to easily distribute software updates to clients. Clients simply connect to the IcePatch2 server and request updates for a particular application. The service automatically checks the version of the client's software and downloads any updated application components in a compressed format to conserve bandwidth. Software patches can be secured using the Glacier2 service, so only authorized clients can download software updates.

Glacier2

Glacier2 is the Ice firewall traversal service: it allows clients and servers to securely communicate through a firewall without compromising security. Client-server traffic is SSL-encrypted using public key certificates and is bidirectional. Glacier2 offers support for mutual authentication as well as secure session management.

IceBridge

IceBridge acts as a bridge between one or more clients and a server and makes every effort to be as transparent as possible.

See Also

- IceGrid
- Glacier2
- IceBox
- IceStorm
- IcePatch2
- IceBridge

Architectural Benefits of Ice

The Ice architecture provides a number of benefits to application developers:

- Object-oriented semantics
  Ice fully preserves the object-oriented paradigm "across the wire." All operation invocations use late binding, so the implementation of an operation is chosen depending on the actual run-time (not static) type of an object.
- Support for synchronous and asynchronous calls
  Ice provides both synchronous and asynchronous operation invocation and dispatch, as well as publish-subscribe messaging via IceStorm. This allows you to choose a communication model according to the needs of your application instead of having to shoe-horn the application to fit a single model.
- Support for multiple interfaces
  With facets, objects can provide multiple, unrelated interfaces while retaining a single object identity across these interfaces. This provides great flexibility, particularly as an application evolves but needs to remain compatible with older, already deployed clients.
- Machine independence
  Clients and servers are shielded form idiosyncrasies of the underlying machine architecture. Issues such as byte ordering and padding are hidden from application code.
- Language independence
  Client and server can be developed independently and in different programming languages. The Slice definition used by both client and server establishes the interface contract between them and is the only thing they need to agree on.
- Implementation independence
  Clients are unaware of how servers implement their objects. This means that the implementation of a server can be changed after clients are deployed, for example, to use a different persistence mechanism or even a different programming language.
- Operating system independence
  The Ice APIs are fully portable, so the same source code compiles and runs under both Windows and Unix.
- Threading support
  The Ice run time is fully threaded and APIs are thread-safe. No effort (beyond synchronizing access to shared data) is required on part of the application developer to develop threaded, high-performance clients and servers.
- Transport independence
  Ice supports TCP/IP, UDP, Bluetooth and iAP. Neither client nor server code are aware of the underlying transport. (The desired transport can be chosen by a configuration parameter.)
- Location and server transparency
  The Ice run time takes care of locating objects and managing the underlying transport mechanism, such as opening and closing connections. Interactions between client and server appear connection-less. Via IceGrid, you can arrange for servers to be started on demand if they are not running at the time a client invokes an operation. Servers can be migrated to different physical addresses without breaking proxies held by clients, and clients are completely unaware how object implementations are distributed over server processes.
- Security
  Communications between client and server can be fully secured with strong encryption over SSL, so applications can use unsecured public networks to communicate securely. Via Glacier2, you can implement secure forwarding of requests through a firewall, with full support for callbacks.

See Also

- Ice Architecture
- Ice Services Overview

Hello World Application

This section presents a very simple (but complete) client and server.

Writing an Ice application involves the following steps:

1. Write a Slice definition and compile it.
2. Write a server and compile it.
3. Write a client and compile it.

If someone else has written the server already and you are only writing a client, you do not need to write the Slice definition, only compile it (and, obviously, you do not need to write the server in that case).

The application described here enables remote printing: a client sends the text to be printed to a server, which in turn sends that text to a printer. For simplicity (and because we do not want to concern ourselves with the idiosyncrasies of print spoolers for various platforms), our printer will simply print to a terminal instead of a real printer. This is no great loss: the purpose of the exercise is to show how a client can communicate with a server; once the thread of control has reached the server application code, that code can of course do anything it likes (including sending the text to a real printer). How to do this is independent of Ice and therefore not relevant here.

> Much of the detail of the source code will remain unexplained for now. The intent is to show you how to get started and give you a feel for what the development environment looks like; we will provide all the detail throughout the remainder of this manual.

Topics

Writing a Slice File

The first step in writing any Ice application is to write a Slice file containing the Slice definitions that are used by the application. For our minimal printing application, we write the following Slice file:

Slice

```
module Demo
{
    interface Printer
    {
        void printString(string s);
    }
}
```

We save this text in a file called `Printer.ice`.

Our Slice definitions consist of the module `Demo` containing a single interface called `Printer`. For now, the interface is very simple and provides only a single operation, called `printString`. The `printString` operation accepts a string as its sole input parameter; the text of that string is what appears on the (possibly remote) printer.

See Also

- The Slice Language

Writing an Ice Application with C++ (C++11)

This page shows how to create an Ice application with C++ using the Ice C++11 mapping.

On this page:

Compiling a Slice Definition for C++

The first step in creating our C++ application is to compile our Slice definition to generate C++ proxies and skeletons. You can compile the definition as follows:

```
slice2cpp Printer.ice
```

The `slice2cpp` compiler produces two C++ source files from this definition, `Printer.h` and `Printer.cpp`.

- `Printer.h`
  The `Printer.h` header file contains C++ type definitions that correspond to the Slice definitions for our `Printer` interface. This header file must be included in both the client and the server source code.
- `Printer.cpp`
  The `Printer.cpp` file contains the source code for our `Printer` interface. The generated source contains type-specific run-time support for both clients and servers. For example, it contains code that marshals parameter data (the string passed to the `printString` operation) on the client side and unmarshals that data on the server side.
  The `Printer.cpp` file must be compiled and linked into both client and server.

Writing and Compiling a Server in C++

The source code for the server takes only a few lines and is shown in full here:

C++

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

class PrinterI : public Printer
{
public:
    virtual void printString(string s, const Ice::Current&) override;
};

void
PrinterI::printString(string s, const Ice::Current&)
{
    cout << s << endl;
}

int
main(int argc, char* argv[])
{
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);
        auto adapter =
ich->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default
-p 10000");
        auto servant = make_shared<PrinterI>();
        adapter->add(servant, Ice::stringToIdentity("SimplePrinter"));
        adapter->activate();
        ich->waitForShutdown();
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        return 1;
    }
    return 0;
}
```

Every Ice source file starts with an include directive for `Ice.h`, which contains the definitions for the Ice run time. We also include `Printer.h`, which was generated by the Slice compiler and contains the C++ definitions for our printer interface, and we import the contents of the `std` and `Demo` namespaces for brevity in the code that follows:

C++

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;
```

Our server implements a single printer servant, of type `PrinterI`. Looking at the generated code in `Printer.h`, we find the following (tidied up a little to get rid of irrelevant detail):

C++

```
namespace Demo
{
    class Printer : public virtual Ice::Object
    {
    public:
        virtual void printString(std::string, const Ice::Current&) = 0;
    };
}
```

The `Printer` skeleton class definition is generated by the Slice compiler. (Note that the `printString` method is pure virtual so the skeleton class cannot be instantiated.) Our servant class inherits from the skeleton class to provide an implementation of the pure virtual `printString` method. (By convention, we use an `I`-suffix to indicate that the class implements an interface.)

C++

```
class PrinterI : public Printer
{
public:
    virtual void printString(string s, const Ice::Current&) override;
};
```

The implementation of the `printString` method is trivial: it simply writes its string argument to `stdout`:

C++

```
void
PrinterI::printString(string s, const Ice::Current&)
{
    cout << s << endl;
}
```

Note that `printString` has a second parameter of type `Ice::Current`. As you can see from the definition of `Printer::printString`, the Slice compiler generates a default argument for this parameter, so we can leave it unused in our implementation. (We will examine the purpose of the `Ice::Current` parameter later.)

What follows is the server main program. Note the general structure of the code:

C++

```
int
main(int argc, char* argv[])
{
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);
        // Server implementation here ...
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        return 1;
    }
    return 0;
}
```

The body of `main` contains a try/catch block, and we start by creating an Ice `CommunicatorHolder` on the stack. We pass `argc` and `argv` to the `CommunicatorHolder` because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.

Next, we have the actual server code:

C++

```
auto adapter =
ich->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default
-p 10000");
        auto servant = make_shared<PrinterI>();
        adapter->add(servant,
icHolder->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ich->waitForShutdown();
```

The code goes through the following steps:

1. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance (through `CommunicatorHolder's` overloaded arrow `operator`). The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
2. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
3. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
4. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.) The server starts to process incoming requests from clients as soon as the adapter is activated.
5. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Assuming that we have the server code in a file called `Server.cpp`, we can compile it as follows:

```
c++ -I. -DICE_CPP11_MAPPING -c Printer.cpp Server.cpp
```

This compiles both our application code and the code that was generated by the Slice compiler. Depending on your platform, you may have to add additional include directives or other options to the compiler; please see the demo programs that ship with Ice for the details.

`-DICE_CPP11_MAPPING` enables the new Ice C++11 mapping

Finally, we need to link the server into an executable:

```
c++ -o server Printer.o Server.o -lIce++11
```

Again, depending on the platform, the actual list of libraries you need to link against may be longer. The demo programs that ship with Ice contain all the detail.

Writing and Compiling a Client in C++

The client code looks very similar to the server. Here it is in full:

C++

```
#include <Ice/Ice.h>
#include <Printer.h>
#include <stdexcept>

using namespace std;
using namespace Demo;

int
main(int argc, char* argv[])
{
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);

auto base = ich->stringToProxy("SimplePrinter:default -p 10000");
        auto printer = Ice::checkedCast<PrinterPrx>(base);
        if(!printer)
        {
            throw std::runtime_error("Invalid proxy");
        }

        printer->printString("Hello World!");
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        return 1;
    }
    return 0;
}
```

Note that the overall code layout is the same as for the server: we include the headers for the Ice run time and the header generated by the Slice compiler, and we use the same try/catch blocks to deal with errors.

The client code does the following:

1. As for the server, we initialize the Ice run time by creating an `Ice::CommunicatorHolder` object, which creates and holds an `Ice::Communicator`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `checkedCast<PrinterPrx>`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy to a `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns a null proxy.
4. We test that the down-cast succeeded and, if not, throw a `runtime_error` that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling and linking the client looks much the same as for the server:

```
c++ -I. -DICE_CPP11_MAPPING -c Printer.cpp Client.cpp
c++ -o client Printer.o Client.o -lIce++11
```

Running Client and Server in C++

To run client and server, we first start the server in a separate window:

```
./server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
./client
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get:

```
Network.cpp:471: Ice::ConnectFailedException:
connect failed: Connection refused
```

See Also

- The Current Object
- IceGrid

Writing an Ice Application with C++ (C++98)

This page shows how to create an Ice application with C++ using the Ice C++98 mapping.

On this page:

Compiling a Slice Definition for C++

The first step in creating our C++ application is to compile our Slice definition to generate C++ proxies and skeletons. You can compile the definition as follows:

```
slice2cpp Printer.ice
```

The `slice2cpp` compiler produces two C++ source files from this definition, `Printer.h` and `Printer.cpp`.

- `Printer.h`
  The `Printer.h` header file contains C++ type definitions that correspond to the Slice definitions for our `Printer` interface. This header file must be included in both the client and the server source code.
- `Printer.cpp`
  The `Printer.cpp` file contains the source code for our `Printer` interface. The generated source contains type-specific run-time support for both clients and servers. For example, it contains code that marshals parameter data (the string passed to the `printString` operation) on the client side and unmarshals that data on the server side.
  The `Printer.cpp` file must be compiled and linked into both client and server.

Writing and Compiling a Server in C++

The source code for the server takes only a few lines and is shown in full here:

C++

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

class PrinterI : public Printer
{
public:
    virtual void printString(const string& s, const Ice::Current&);
};

void
PrinterI::
printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}

int
main(int argc, char* argv[])
{
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);
        Ice::ObjectAdapterPtr adapter =

ich->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default
-p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ich->waitForShutdown();
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        return 1;
    }
    return 0;
}
```

Every Ice source file starts with an include directive for `Ice.h`, which contains the definitions for the Ice run time. We also include `Printer.h`, which was generated by the Slice compiler and contains the C++ definitions for our printer interface, and we import the contents of the `std` and `Demo` namespaces for brevity in the code that follows:

C++

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;
```

Our server implements a single printer servant, of type `PrinterI`. Looking at the generated code in `Printer.h`, we find the following (tidied up a little to get rid of irrelevant detail):

C++

```
namespace Demo
{
    class Printer : virtual public Ice::Object
    {
    public:
        virtual void printString(const std::string&, const Ice::Current&
= Ice::emptyCurrent) = 0;
    };
}
```

The `Printer` skeleton class definition is generated by the Slice compiler. (Note that the `printString` method is pure virtual so the skeleton class cannot be instantiated.) Our servant class inherits from the skeleton class to provide an implementation of the pure virtual `printString` method. (By convention, we use an `I`-suffix to indicate that the class implements an interface.)

C++

```
class PrinterI : public Printer
{
public:
    virtual void printString(const string& s, const Ice::Current&);
};
```

The implementation of the `printString` method is trivial: it simply writes its string argument to `stdout`:

C++

```
void
PrinterI::printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}
```

Note that `printString` has a second parameter of type `Ice::Current`. As you can see from the definition of `Printer::printString`, the Slice compiler generates a default argument for this parameter, so we can leave it unused in our implementation. (We will examine the purpose of the `Ice::Current` parameter later.)

What follows is the server main program. Note the general structure of the code:

C++

```
int
main(int argc, char* argv[])
{
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);
        // Server implementation here...
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        return 1;
    }
    return 0;
}
```

The body of `main` contains a try/catch block, and we start by creating an Ice Communicator holder on the stack. We pass `argc` and `argv` to the `CommunicatorHolder` because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments. `CommunicatorHolder` is a a RAII-helper class, which creates and holds an `Ice::Communicator` object. The primary purpose of this holder object is to call `destroy` on the communicator when the holder goes out of scope.

> Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

Next, we have the actual server code:

C++

```
        Ice::ObjectAdapterPtr adapter =

ich->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default
-p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ich->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ich->waitForShutdown();
```

The code goes through the following steps:

1. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance (through `CommunicatorHolder's` overloaded `operator->()`). The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
2. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
3. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)

4. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.) The server starts to process incoming requests from clients as soon as the adapter is activated.
5. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Assuming that we have the server code in a file called `Server.cpp`, we can compile it as follows:

```
c++ -I. -c Printer.cpp Server.cpp
```

This compiles both our application code and the code that was generated by the Slice compiler. Depending on your platform, you may have to add additional include directives or other options to the compiler; please see the demo programs that ship with Ice for the details.

Finally, we need to link the server into an executable:

```
c++ -o server Printer.o Server.o -lIce
```

Again, depending on the platform, the actual list of libraries you need to link against may be longer. The demo programs that ship with Ice contain all the detail.

Writing and Compiling a Client in C++

The client code looks very similar to the server. Here it is in full:

C++

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

int
main(int argc, char* argv[])
{
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);
        Ice::ObjectPrx base = ich->stringToProxy("SimplePrinter:default
 -p 10000");
        PrinterPrx printer = PrinterPrx::checkedCast(base);
        if(!printer)
        {
            throw "Invalid proxy";
        }
        printer->printString("Hello World!");
    }
    catch(const std::exception& ex)
    {
        cerr << ex.what() << endl;
        return 1;
    }
    return 0;
}
```

Note that the overall code layout is the same as for the server: we include the headers for the Ice run time and the header generated by the Slice compiler, and we use the same `try` block and `catch` handlers to deal with errors.

The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by creating a `Ice::CommunicatorHolder`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrx::checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy to a `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns a null proxy.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling and linking the client looks much the same as for the server:

```
c++ -I. -I$ICE_HOME/include -c Printer.cpp Client.cpp
c++ -o client Printer.o Client.o -lIce
```

Running Client and Server in C++

To run client and server, we first start the server in a separate window:

```
./server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
./client
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get:

```
Network.cpp:471: Ice::ConnectFailedException:
connect failed: Connection refused
```

See Also

- Client-Side Slice-to-C++98 Mapping
- Server-Side Slice-to-C++98 Mapping
- The Current Object
- IceGrid

Writing an Ice Application with C-Sharp

This page shows how to create an Ice application with C#.

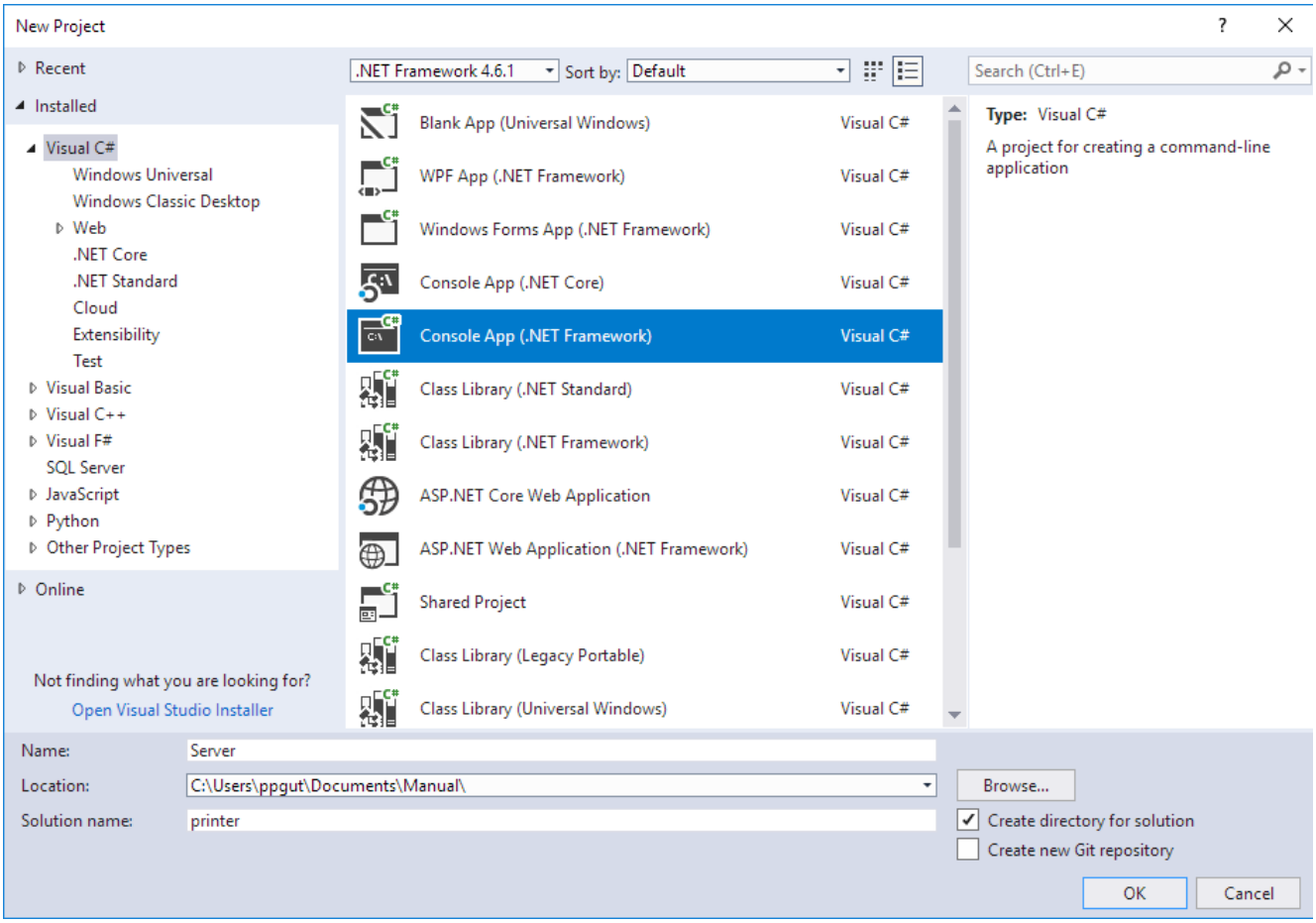
On this page:

Create Projects for your Client and Server Applications

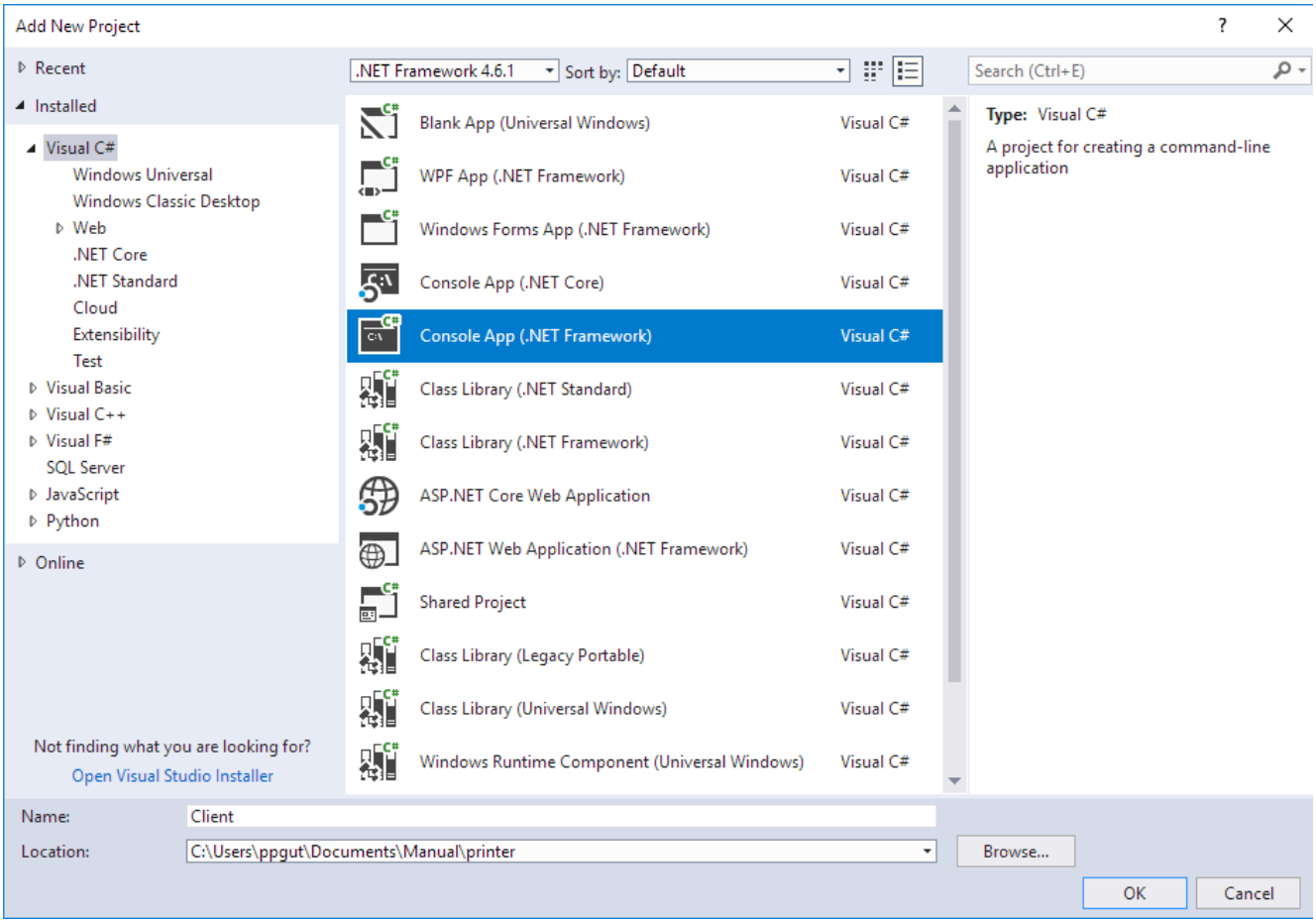
We create two projects, one for the Server application and one for the Client application. These are regular Console projects with very little Ice-specific additions.

.NET Framework 4.5 with Visual Studio.NET Core 2.0 SDK

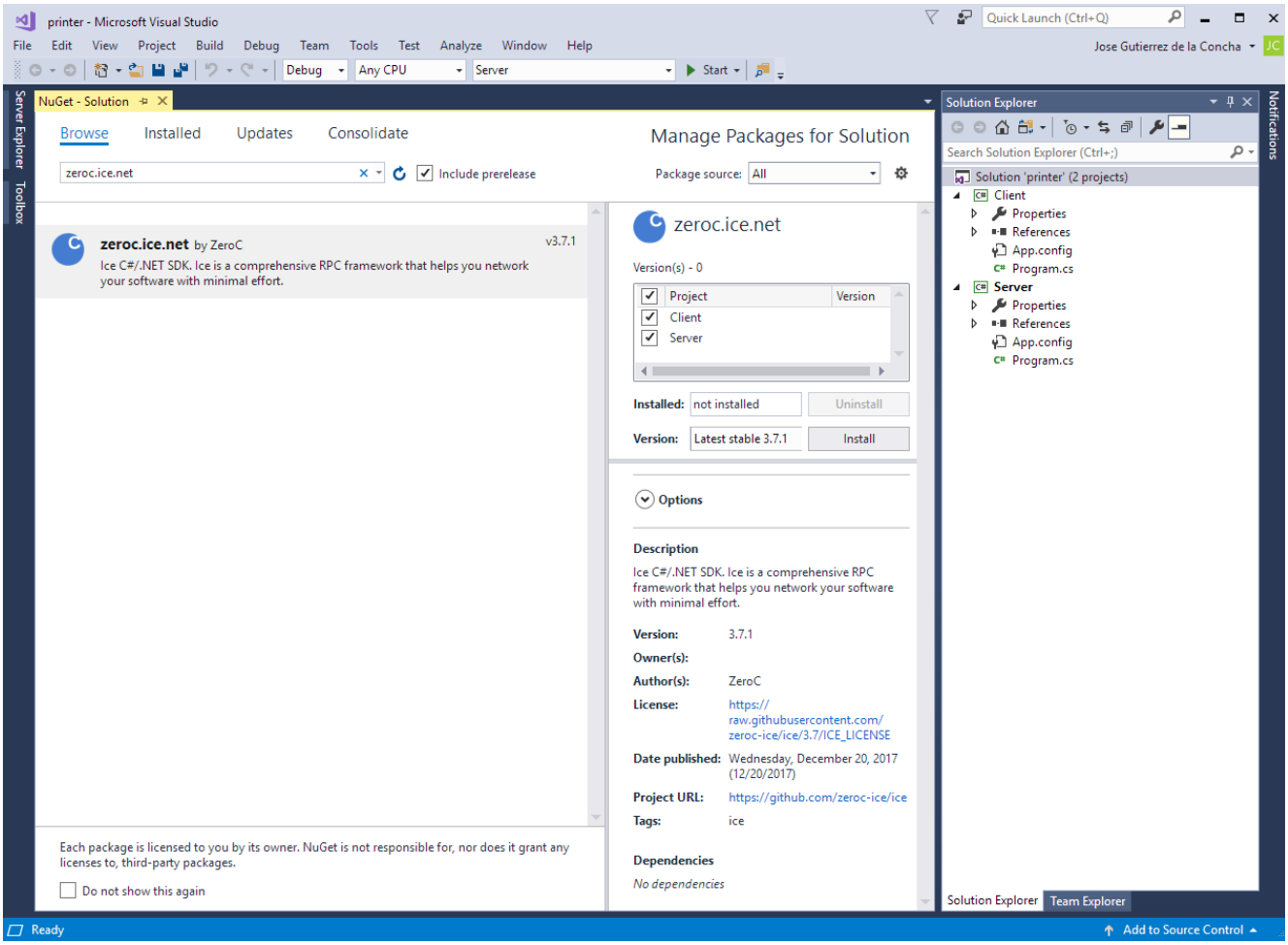
Open Visual Studio and create a new Console Application



Create the client project using "File > Add > New Project..."



Add the `zeroc.icebuilder.msbuild` and `zeroc.ice.net` NuGet package to the projects with the NuGet Package Manager, found in "Tools > NuGet Package Manager > Manage NuGet Packages for Solution...".



Open a new Command Prompt a run the following command to create the server and client projects:

```
dotnet new console -o Server
```

This generates a new .NET Core console application project in the `Server` directory.

Then add references to the `zeroc.icebuilder.msbuild` and `zeroc.ice.net` NuGet packages to this project:

```
dotnet add Server package zeroc.icebuilder.msbuild
dotnet add Server package zeroc.ice.net
```

Finally, repeat these steps for the client project:

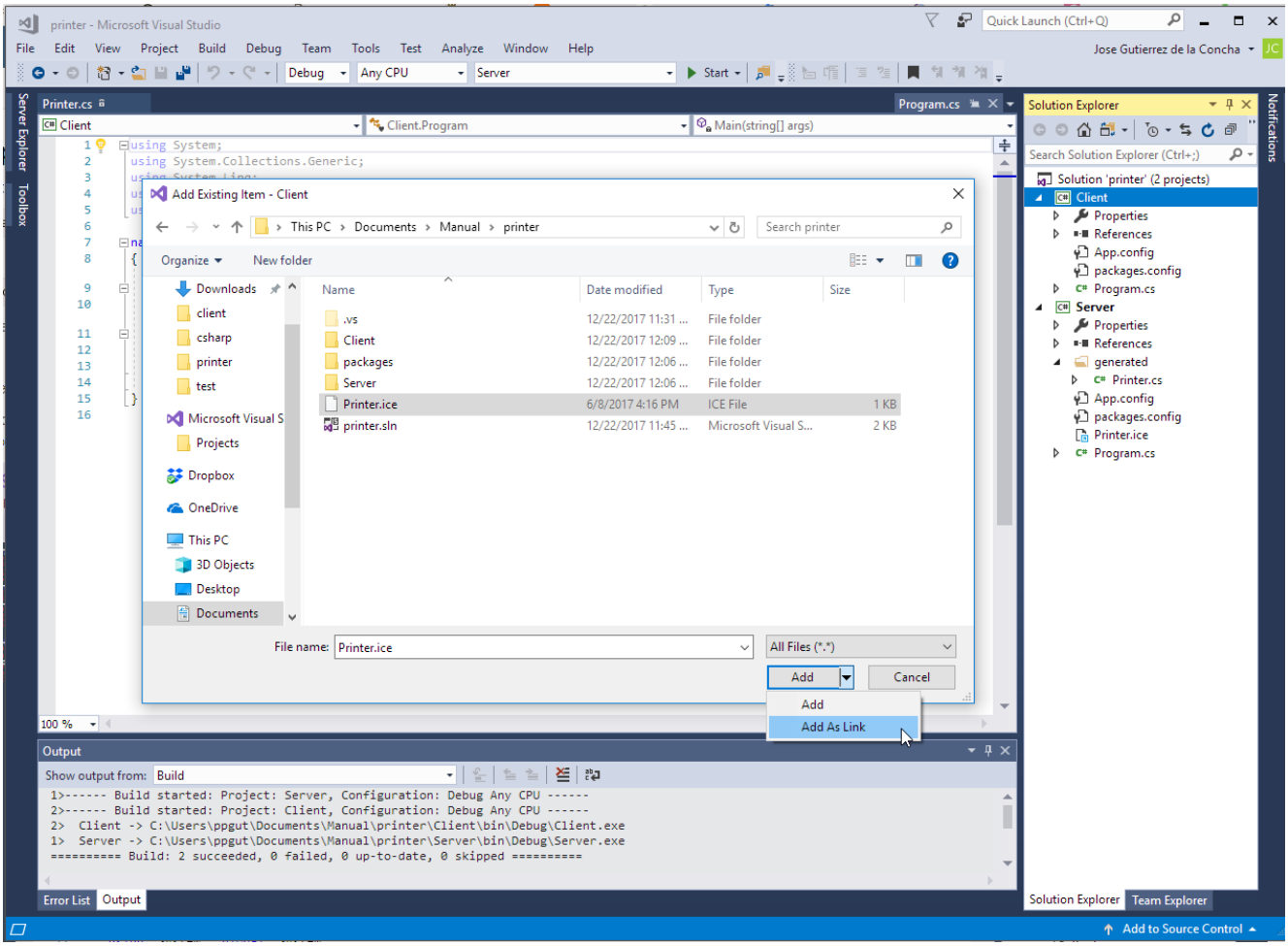
```
dotnet new console -o Client
dotnet add Client package zeroc.icebuilder.msbuild
dotnet add Client package zeroc.ice.net
```

Compile your Slice File

The next step is to add the Slice file (`Printer.ice`) created earlier to each project, and then compile this Slice file.

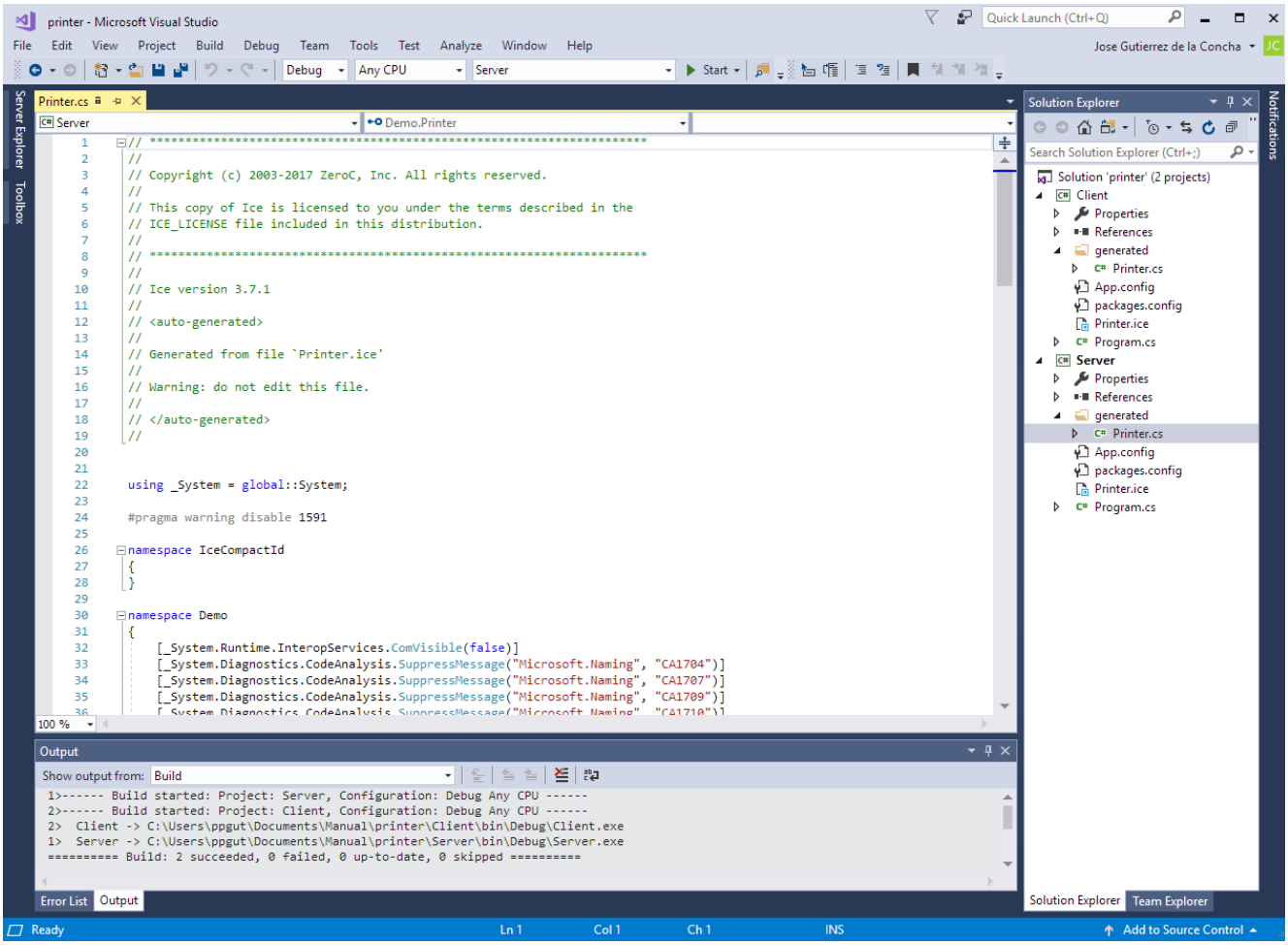
.NET Framework 4.5 with Visual Studio .NET Core 2.0 SDK

Open the "Project > Add Existing Item" dialog and add `Printer.ice` to your Project:



If the Ice Builder for Visual Studio is installed, it immediately generates the file `generated\Printer.cs` from `Printer.ice` unless you disabled automatic building by the Ice Builder.

If you have automatic building disabled, select `Build` to build your project. The build generates `generated\Printer.cs` from `Printer.ice` (using Ice Builder) and then compiles both `generated\Printer.cs` and the default no-op `Program.cs`.



Ice Builder invokes the Slice to C# compiler (`slice2cs`) to compile Slice files into C# files.

Add a Slice item that includes `Printer.ice` to your two projects. The code below shows the client project:

Client.csproj

```
<?xml version="1.0" encoding="utf-8"?>
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>netcoreapp2.0</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <SliceCompile Include="../Printer.ice" />
        <PackageReference Include="zeroc.ice.net" Version="3.7.1" />
        <PackageReference Include="zeroc.icebuilder.msbuild"
Version="5.0.3" />
    </ItemGroup>
</Project>
```

When building the project, the `SliceCompile` task (imported automatically from the `zeroc.icebuilder.msbuild` NuGet package) compiles `Printer.ice` into `generated/Printer.cs` using the Slice to C# compiler, slice2cs.

Use the following command to build the projects:

```
dotnet build Server
dotnet build Client
```

Write and Compile your Server

To implement our `Printer` interface, we must create a servant class. By convention, a servant class uses the name of its interface with an `I`-suffix, so our servant class is called `PrinterI` and we will place it into the default C# source file Program.cs:

C#

```
using System;

namespace Server
{
    public class PrinterI : Demo.PrinterDisp_
    {
        public override void printString(string s, Ice.Current current)
        {
            Console.WriteLine(s);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

The `PrinterI` class inherits from a base class called `PrinterDisp_`, which is generated by the `slice2cs` compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code follows in `Program.cs` and is shown in full here:

C#

```
using System;

namespace Server
{
    public class PrinterI : Demo.PrinterDisp_
    {
        public override void printString(string s, Ice.Current current)
        {
            Console.WriteLine(s);
        }
    }

    public class Program
    {
        public static int Main(string[] args)
        {
            try
            {
                using(Ice.Communicator communicator =
Ice.Util.initialize(ref args))
                {
                    var adapter =

communicator.createObjectAdapterWithEndpoints("SimplePrinterAdapter",
"default -h localhost -p 10000");
                    adapter.add(new PrinterI(),
Ice.Util.stringToIdentity("SimplePrinter"));
                    adapter.activate();
                    communicator.waitForShutdown();
                }
            }
            catch(Exception e)
            {
                Console.Error.WriteLine(e);
                return 1;
            }
            return 0;
        }
    }
}
```

The body of `Main` contains a `try` block in which we place all the server code, followed by a `catch` block. The catch block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to `Main`, which prints the exception and then returns failure to the operating system.

The `Ice.Communicator` object implements `IDisposable`, which allows us to use the `using` statement for the initialization of the `Ice.Communicator` object. This ensures the communicator `destroy` method is called when the `using` block goes out of scope. Doing this is essential in order to correctly finalize the Ice run time.

The body of our `try` block contains the actual server code.

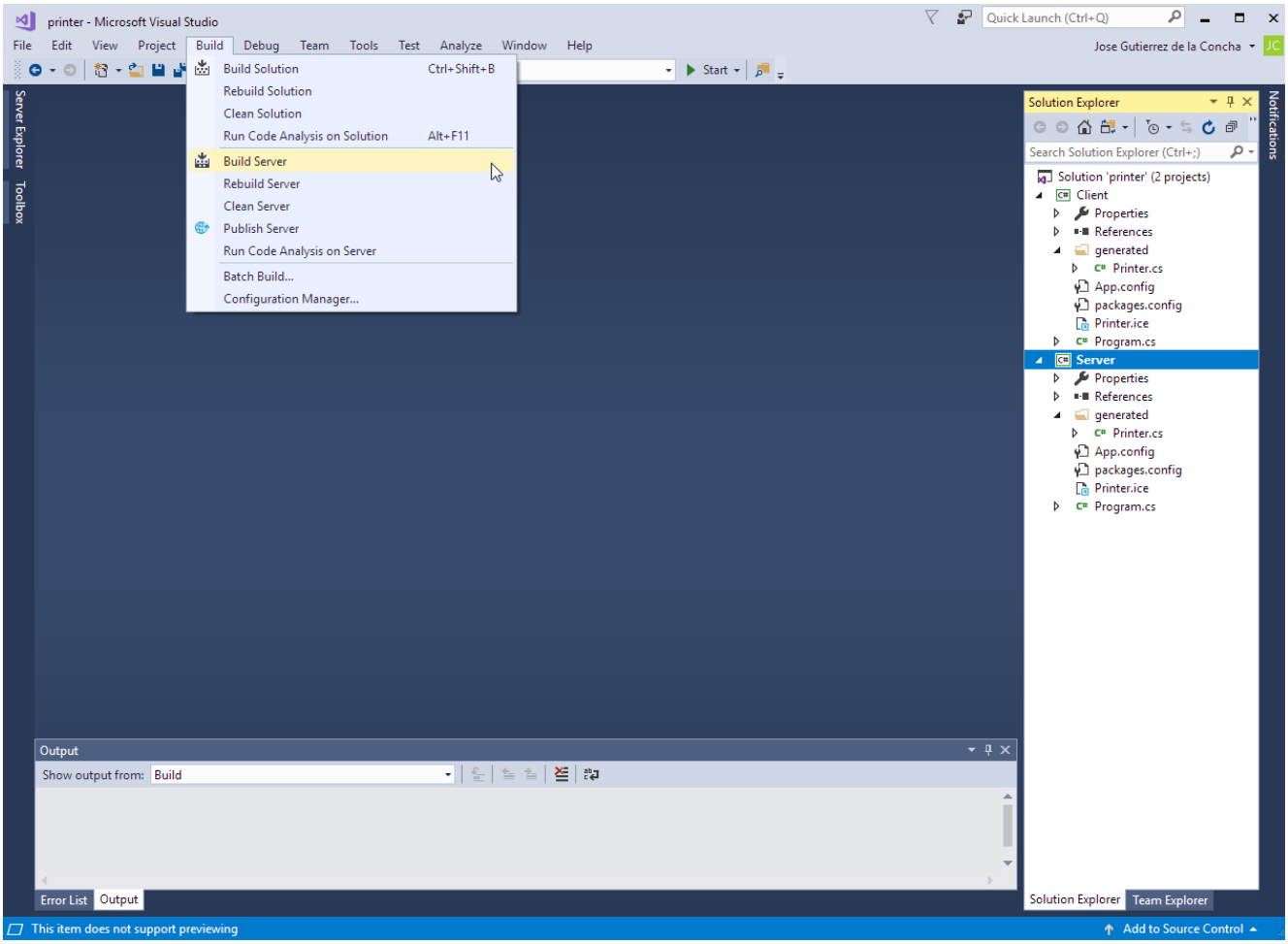
The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

We can compile the server code as follows:

.NET Framework 4.5 with Visual Studio.NET Core 2.0 SDK

Build the server project using "Build > Builder Server"



Build the server project using the dotnet `build` command:

```
cd Server
dotnet build
```

Write and Compile your Client

The client code, in `Client/Program.cs`, looks very similar to the server.

Here it is in full:

C#

```
using Demo;
using System;

namespace Client
{
    public class Program
    {
        public static int Main(string[] args)
        {
            try
            {
                using(Ice.Communicator communicator =
Ice.Util.initialize(ref args))
                {
                    var obj =
communicator.stringToProxy("SimplePrinter:default -h localhost -p
10000");
                    var printer = PrinterPrxHelper.checkedCast(obj);
                    if(printer == null)
                    {
                        throw new ApplicationException("Invalid proxy");
                    }

                    printer.printString("Hello World!");
                }
            }
            catch(Exception e)
            {
                Console.Error.WriteLine(e);
                return 1;
            }
            return 0;
        }
    }
}
```

Note that the overall code layout is the same as for the server: we use the same `try` and `catch` blocks to deal with errors. The code in the `try` block does the following:

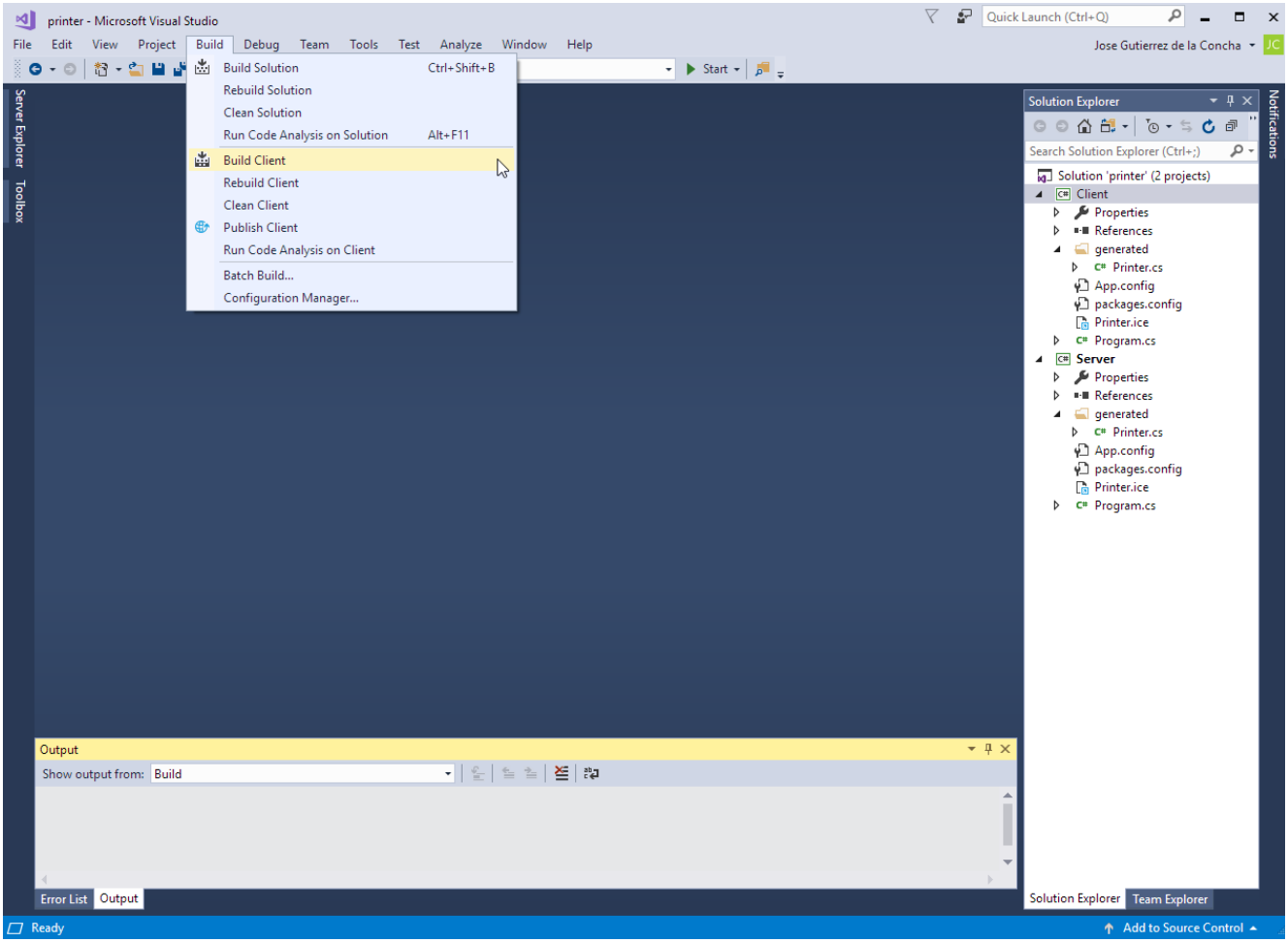
1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize` within the `using` statement
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.
3. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.

5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

The client's project is just like the server's project shown earlier.

.NET Framework 4.5 with Visual Studio.NET Core 2.0 SDK

Build the client project using "Build > Builder Client"



Build the client project using dotnet `build` command:

```
cd Client
dotnet build
```

Run your Client and Server

To run client and server, we first start the server in a separate window:

.NET Framework 4.5.NET Core 2.0

```
server
```

```
cd Server
dotnet run
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

.NET Framework 4.5.NET Core 2.0

```
client
```

```
cd Client
dotnet run
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we just interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice.ConnectionRefusedException
    error = 0
   at Ice.ObjectPrxHelperBase.ice_isA(String id, OptionalContext
context) in
C:\Users\vagrant\workspace\ice-dist\netcore\dist-utils\build\ice-netcor
e\builds\ice-VC141\csharp\src\Ice\Proxy.cs:line 887
   at Demo.PrinterPrxHelper.checkedCast(ObjectPrx b) in
D:\3.7\ice-demos\csharp\Manual\net45\printer\Client\generated\Printer.c
s:line 196
   at Client.Program.Main(String[] args) in
D:\3.7\ice-demos\csharp\Manual\net45\printer\Client\Program.cs:line 18
Caused by: System.Net.Sockets.SocketException: No connection could be
made because the target machine actively refused it
```

See Also

- Client-Side Slice-to-C-Sharp Mapping
- Server-Side Slice-to-C-Sharp Mapping
- The Current Object
- IceGrid

Writing an Ice Application with Java

This page shows how to create an Ice application with Java.

On this page:

Create Projects for your Client and Server Applications

We will use Gradle to create our application projects. You must install Gradle before continuing with this tutorial.

Open a new Command Prompt and run the following commands to generate a new project:

```
mkdir printer
cd printer
gradle init
```

For this demo we're going to use a project with two sub-projects to build the Client and Server applications. The requirements for our sub-projects are the same so we'll do all the setup in the `subprojects` block of the root project, which applies to all sub-projects. Edit the generated `build.gradle` file to look like the one below:

build.gradle

```
//
// Install the gradle Ice Builder plug-in from the plug-in portal
//
plugins {
    id 'com.zeroc.gradle.ice-builder.slice' version '1.4.5' apply false
}

subprojects {
    //
    // Apply Java and Ice Builder plug-ins to all sub-projects
    //
    apply plugin: 'java'
    apply plugin: 'com.zeroc.gradle.ice-builder.slice'

    //
    // Both Client and Server projects share the Printer.ice Slice
definitions
    //
    slice {
        java {
            files = [file("../Printer.ice")]
        }
    }

    //
```

```
    // Use Ice JAR files from maven central repository
    //
    repositories {
        mavenCentral()
    }

    //
    // Both Client and Server depend only on Ice JAR
    //
    dependencies {
        compile 'com.zeroc:ice:3.7.1'
    }

    //
    // Create a JAR file with the appropriate Main-Class and Class-Path
attributes
    //
    jar {
        manifest {
            attributes(
                "Main-Class" : project.name.capitalize(),
                "Class-Path": configurations.runtime.resolve().collect {
it.toURI() }.join(' ')
            )
```

```
        }
    }
}
```

We must also edit the generated `settings.gradle` to define our sub-projects:

settings.gradle

```
rootProject.name = 'printer'
include 'client'
include 'server'
```

Finally we need to create the directories for client and server projects:

```
mkdir client
mkdir server
```

Compiling a Slice Definition for Java

The next step is to add the Slice file (`Printer.ice`), and then compile this Slice file. When building the project, the `sliceCompile` task (added automatically by the Ice Builder plug-in) compiles `Printer.ice` and places the generated code into `build/generated-src` using the Slice to Java compiler, `slice2java`.

Writing and Compiling a Server in Java

To implement our `Printer` interface, we must create a servant class. By convention, a servant class uses the name of its interface with an `I`-suffix, so our servant class is called `PrinterI` and placed into a source file `server/src/main/java/PrinterI.java`:

server/src/main/java/PrinterI.java

```
public class PrinterI implements Demo.Printer
{
    public void printString(String s, com.zeroc.Ice.Current current)
    {
        System.out.println(s);
    }
}
```

The `PrinterI` class implements the interface `Printer`, which is generated by the `slice2java` compiler. The interface defines a `printString` method that accepts a string for the printer to print and a parameter of type `Current`. (For now we will ignore the `Current` parameter.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code is in a source file called `server/src/main/java/Server.java`, shown in full here:

server/src/main/java/Server.java

```
public class Server
{
    public static void main(String[] args)
    {
        try(com.zeroc.Ice.Communicator communicator
= com.zeroc.Ice.Util.initialize(args))
        {
            com.zeroc.Ice.ObjectAdapter adapter =
communicator.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "
default -p 10000");
            com.zeroc.Ice.Object object = new PrinterI();
            adapter.add(object,
com.zeroc.Ice.Util.stringToIdentity("SimplePrinter"));
            adapter.activate();
            communicator.waitForShutdown();
        }
    }
}
```

The body of `main` contains a `try-with-resources` block in which we place all the server code. The `Communicator` object implements `java.lang.AutoCloseable`, which allows us to use the `try-with-resources` statement for the initialization of the `Communicator` object. This ensures the communicator `destroy` method is called when the `try` block goes out of scope. Doing this is essential in order to correctly finalize the Ice run time.

A communicator starts a number of non-background threads. Destroying the communicator terminates all these threads.

The body of our `try` block contains the actual server code. The code goes through the following steps:

1. We initialize the Ice run time by calling `com.zeroc.Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns a `Communicator` reference, which is the main object in the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

We can compile the server code as follows:

```
gradlew :server:build
```

Writing and Compiling a Client in Java

The client code, in `client/src/main/java/Client.java`, looks very similar to the server. Here it is in full:

client/src/main/java/Client.java

```
public class Client
{
    public static void main(String[] args)
    {
        try(com.zeroc.Ice.Communicator communicator
= com.zeroc.Ice.Util.initialize(args))
        {
            com.zeroc.Ice.ObjectPrx base = communicator.stringToProxy("
SimplePrinter:default -p 10000");
            Demo.PrinterPrx printer = Demo.PrinterPrx.checkedCast(base);
            if(printer == null)
            {
                throw new Error("Invalid proxy");
            }
            printer.printString("Hello World!");
        }
    }
}
```

Note that the overall code layout is the same as for the server: we use the same `try` and `catch` blocks to deal with errors. The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `com.zeroc.Ice.Util.initialize` within the Java `try-with-resources` statement.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `com.zeroc.Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling the client looks much the same as for the server:

```
gradlew :client:build
```

Running Client and Server in Java

To run client and server, we first start the server in a separate window:

```
java -jar server/build/libs/server.jar
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
java -jar client/builds/libs/client.jar
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
com.zeroc.Ice.ConnectionRefusedException
       error = 0
    at ...
    at Client.run(Client.java:65)
Caused by: java.net.ConnectException: Connection refused
         ...
```

See Also

- Client-Side Slice-to-Java Mapping
- Server-Side Slice-to-Java Mapping
- The Current Object
- IceGrid

Writing an Ice Application with Java Compat

This page shows how to create an Ice application with Java Compat.

On this page:

Create Projects for your Client and Server Applications

We will use Gradle to create our application projects. You must install Gradle before continuing with this tutorial.

Open a new Command Prompt and run the following commands to generate a new project:

```
mkdir printer
cd printer
gradle init
```

For this demo we're going to use a project with two sub-projects to build the Client and Server applications. The requirements for our sub-projects are the same so we'll do all the setup in the `subprojects` block of the root project, which applies to all sub-projects. Edit the generated `build.gradle` file to look like the one below:

build.gradle

```
//
// Install the gradle Ice Builder plug-in from the plug-in portal
//
plugins {
    id 'com.zeroc.gradle.ice-builder.slice' version '1.4.5' apply false
}

subprojects {
    //
    // Apply Java and Ice Builder plug-ins to all sub-projects
    //
    apply plugin: 'java'
    apply plugin: 'com.zeroc.gradle.ice-builder.slice'

    //
    // Both Client and Server projects share the Printer.ice Slice
definitions
    //
    slice {
        java {
            files = [file("../Printer.ice")]
            compat = true
        }
    }
```

```
    //
    // Use Ice JAR files from maven central repository
    //
    repositories {
        mavenCentral()
    }

    //
    // Both Client and Server depend only on Ice JAR
    //
    dependencies {
        compile 'com.zeroc:ice-compat:3.7.1'
    }

    //
    // Create a JAR file with the appropriate Main-Class and Class-Path
attributes
    //
    jar {
        manifest {
            attributes(
                "Main-Class" : project.name.capitalize(),
                "Class-Path": configurations.runtime.resolve().collect {
it.toURI() }.join(' ')
            )
```

```
        }
    }
}
```

We must also edit the generated `settings.gradle` to define our sub-projects:

settings.gradle

```
rootProject.name = 'printer'
include 'client'
include 'server'
```

Finally we need to create the directories for client and server projects:

```
mkdir client
mkdir server
```

Compiling a Slice Definition for Java

The next step is to add the Slice file (`Printer.ice`), and then compile this Slice file. When building the project, the `sliceCompile` task (added automatically by the Ice Builder plug-in) compiles `Printer.ice` and places the generated code into `build/generated-src` using the Slice to Java compiler, `slice2java`.

Writing and Compiling a Server in Java

To implement our `Printer` interface, we must create a servant class. By convention, a servant class uses the name of its interface with an `I`-suffix, so our servant class is called `PrinterI` and placed into a source file server/src/main/java/PrinterI.java:

server/src/main/java/PrinterI.java

```
public class PrinterI extends Demo._PrinterDisp
{
    public void printString(String s, Ice.Current current)
    {
        System.out.println(s);
    }
}
```

The `PrinterI` class extends the class `_PrinterDisp`, which is generated by the `slice2java` compiler. The class defines a `printString` method that accepts a string for the printer to print and a parameter of type `Current`. (For now we will ignore the `Current` parameter.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code is in a source file called `server/src/main/java/Server.java`, shown in full here:

server/src/main/java/Server.java

```
public class Server
{
    public static void main(String[] args)
    {
        try(Ice.Communicator communicator = Ice.Util.initialize(args))
        {
            Ice.ObjectAdapter adapter =
communicator.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "
default -p 10000");
            Ice.Object object = new PrinterI();
            adapter.add(object,
Ice.Util.stringToIdentity("SimplePrinter"));
            adapter.activate();
            communicator.waitForShutdown();
        }
    }
}
```

The body of `main` contains a `try-with-resources` block in which we place all the server code.The `Communicator` object implements `java.lang.AutoCloseable`, which allows us to use the `try-with-resources` statement for the initialization of the `Communicator` object. This ensures the communicator `destroy` method is called when the `try` block goes out of scope. Doing this is essential in order to correctly finalize the Ice run time.

A communicator starts a number of non-background threads. Destroying the communicator terminates all these threads.

The body of our `try` block contains the actual server code.

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns a `Communicator` reference, which is the main object in the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

We can compile the server code as follows:

```
gradlew :server:build
```

Writing and Compiling a Client in Java

The client code, in `client/src/main/java/Client.java`, looks very similar to the server. Here it is in full:

client/src/main/java/Client.java

```
public class Client
{
    public static void main(String[] args)
    {
        try(Ice.Communicator communicator = Ice.Util.initialize(args))
        {
            Ice.ObjectPrx base = communicator.stringToProxy("SimplePrin
ter:default -p 10000");
            Demo.PrinterPrx printer
= Demo.PrinterPrxHelper.checkedCast(base);
            if(printer == null)
            {
                throw new Error("Invalid proxy");
            }
            printer.printString("Hello World!");
        }
    }
}
```

Note that the overall code layout is the same as for the server: we use the same `try` and `catch` blocks to deal with errors. The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize` within the Java `try-with-resources` statement.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling the client looks much the same as for the server:

```
gradlew :client:build
```

Running Client and Server in Java

To run client and server, we first start the server in a separate window:

```
java -jar server/build/libs/server.jar
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
java -jar client/build/libs/client.jar
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice.ConnectionRefusedException
       error = 0
    at ...
    at Client.run(Client.java:65)
Caused by: java.net.ConnectException: Connection refused
         ...
```

See Also

- Client-Side Slice-to-Java Mapping
- Server-Side Slice-to-Java Mapping
- The Current Object
- IceGrid

Writing an Ice Application with JavaScript

This page shows how to create an Ice client application with JavaScript.

On this page:

Compiling a Slice Definition for JavaScript

The first step in creating our JavaScript application is to compile our Slice definition to generate JavaScript proxies. You can compile the definition as follows:

```
slice2js Printer.ice
```

The `slice2js` compiler produces a single source file, `Printer.js`, from this definition. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Using Ice with NodeJS

The language mapping is the same whether you're writing applications for NodeJS or a browser, but the code style is different enough that we describe the two platforms separately.

Writing a NodeJS Client

The client code, in `Client.js`, is shown below in full:

JavaScript

```
const Ice = require("ice").Ice;
const Demo = require("./Printer").Demo;

(async function()
{
    let ic;
    try
    {
        ic = Ice.initialize();
        const base = ic.stringToProxy("SimplePrinter:default -p 10000");
        const printer = await Demo.PrinterPrx.checkedCast(base);
        if(printer)
        {
            await printer.printString("Hello World!");
        }
        else
        {
            console.log("Invalid proxy");
        }
    }
    catch(ex)
    {
        console.log(ex.toString());
        process.exitCode = 1;
    }
    finally
    {
        if(ic)
        {
            await ic.destroy();
        }
    }
}());
```

The program begins with `require` statements that assign modules from the Ice run time and the generated code to convenient local variables. (These statements are necessary for use with NodeJS. Browser applications would omit these statements and load the modules a different way.)

The program then defines an asynchronous function, which allows us to use the `await` keyword in our code when making proxy invocations. Here are the notable aspects of this code:

1. The body of the function begins by calling `Ice.initialize` to initialize the Ice run time. The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we

need to do a down-cast by calling `Demo.PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo::PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `null`.

4. The `checkedCast` function involves a remote invocation to the server, which means this function has asynchronous semantics and therefore it returns a new promise object. We apply the `await` keyword to the promise to wait for the call to complete.
5. If `checkedCast` returns a non-null value, we now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal. Again, `printString` is a remote invocation, and it returns a promise that we await.
6. The `finally` block is executed after the `try` block has completed, whether or not it completes successfully. If we successfully created a communicator in the `try` block, we destroy it here. Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results. The `destroy` function has asynchronous semantics, so we await it to ensure no subsequent code is executed until `destroy` completes.

Running the NodeJS Client

The server must be started before the client. Since Ice for JavaScript does not currently include a complete server-side implementation, we need to use a server from another language mapping. In this case, we will use the C++ server:

```
server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
node Client.js
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice::ConnectionRefusedException
    ice_cause: "Error: connect ECONNREFUSED"
    error: "ECONNREFUSED"
```

Note that, to successfully run the client, NodeJS must be able to locate the Ice for JavaScript modules. See the Ice for JavaScript installation instructions for more information.

Using Ice in a Browser

The client code, in `Client.js`, is shown below in full:

JavaScript

```
(function(){

const communicator = Ice.initialize();

async function printString()
{
    try
    {
        setState(State.Busy);
```

```
        const hostname = document.location.hostname || "127.0.0.1";
        const proxy = communicator.stringToProxy(`SimplePrinter:ws -h
${hostname} -p 10000`);

        const printer = await Demo.PrinterPrx.checkedCast(proxy);
        if(printer)
        {
            await printer.printString("Hello World!");
        }
        else
        {
            $("#output").val("Invalid proxy");
        }
    }
    catch(ex)
    {
        $("#output").val(ex.toString());
    }
    finally
    {
        setState(State.Idle);
    }
}

const State =
{
    Idle: 0,
    Busy: 1
};

let state;

function setState(newState)
{
    switch(newState)
    {
        case State.Idle:
        {
            // Hide the progress indicator.
            $("#progress").hide();
            $("body").removeClass("waiting");
            // Enable the button
            $("#print").removeClass("disabled").click(printString);
            break;
        }
        case State.Busy:
        {
            // Clear any previous error messages.
            $("#output").val("");
```

```
            // Disable buttons.
            $("#print").addClass("disabled").off("click");
            // Display the progress indicator and set the wait cursor.
            $("#progress").show();
            $("body").addClass("waiting");
            break;
        }
    }
    state = newState;
}
```

```
setState(State.Idle);
}());
```

Here are the notable aspects of this code:

1. The program begins by calling `Ice.initialize` to initialize the Ice run time. The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
2. Next the program defines the asynchronous function `printString`, which serves as the callback function for a UI button press. The `async` qualifier allows us to use the `await` keyword when making proxy invocations.
3. The code uses a simple state machine to manage the UI elements. Before making a remote invocation, the function enters the "busy" state to update the UI elements.
4. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:ws -h` *`hostname`* `-p 10000"`, where *`hostname`* is the document location. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
5. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo.PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo::PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `null`.
6. The `checkedCast` function involves a remote invocation to the server, which means this function has asynchronous semantics and therefore it returns a new promise object. We apply the `await` keyword to the promise to wait for the call to complete.
7. If `checkedCast` returns a non-null value, we now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal. Again, `printString` is a remote invocation, and it returns a promise that we await.
8. The `finally` block is executed after the `try` block has completed, whether or not it completes successfully, in order to reset the program's state to "idle".

Here are some snippets from the corresponding HTML code:

HTML

```
<script type="text/javascript" src="Ice.js">
<script type="text/javascript" src="Printer.js">
<script type="text/javascript" src="Client.js">
...
<!-- UI elements -->
<section role="main" id="body">
    <div class="row">
        <div class="large-12 medium-12 columns">
            <form>
                <div class="row">
                    <div class="small-12 columns">
                        <a href="#" class="button small"
id="print">Print String</a>
                    </div>
                </div>
                <div class="row">
                    <div class="small-12 columns">
                        <textarea id="output" readonly></textarea>
                    </div>
                </div>
                <div id="progress" class="row hide">
                    <div class="small-12 columns left">
                        <div class="inline left icon"></div>
                        <div class="text">Sending Request...</div>
                    </div>
                </div>
            </form>
        </div>
    </div>
</section>
```

The three `script` elements load the Ice run time, the generated code, and the application code, respectively.

A similar example can be found in `js/Ice/minimal` in the `ice-demos` repository.

See Also

- Client-Side Slice-to-JavaScript Mapping
- IceGrid

Writing an Ice Application with MATLAB

This page shows how to create an Ice client application with MATLAB.

On this page:

Compiling a Slice Definition for MATLAB

The first step in creating our MATLAB application is to compile our Slice definition to generate MATLAB proxies. You can compile the definition as follows:

```
slice2matlab Printer.ice
```

The `slice2matlab` compiler produces a single source file, `Printer.m`, from this definition. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Client in MATLAB

The client code, in `client.m`, is shown below in full:

MATLAB

```
function client()
    if ~libisloaded('ice')
        loadlibrary('ice', @iceproto)
    end
    communicator = Ice.initialize();
    cleanup = onCleanup(@() communicator.destroy());
    base = communicator.stringToProxy('SimplePrinter:default -h
localhost -p 10000');
    printer = Demo.PrinterPrx.checkedCast(base);
    if isempty(printer)
        throw(MException('Client:RuntimeError', 'Invalid proxy'));
    end

    printer.printString('Hello World!');
end
```

The function goes through the following steps:

1. We check whether the Ice library is loaded and call `loadlibrary` if necessary. The `'ice'` argument is the name of the Ice for MATLAB library, while `@iceproto` denotes a prototype file. This file, `iceproto.m`, is included in the Ice for MATLAB distribution. Using a prototype file is simpler and more efficient than a header file because it avoids the need for MATLAB to run a C compiler to preprocess a header file.
2. We initialize the Ice run time by calling `Ice.initialize`. The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
3. We register a "clean up" function that will be called when the `cleanup` variable is reclaimed and ensures that the communicator is destroyed before the program terminates. Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.
4. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with

the string `'SimplePrinter:default -p 10000'`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)

5. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo.PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo.PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns an empty array.
6. We test that the down-cast succeeded and, if not, throw an exception that terminates the client.
7. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `'Hello World!'` string. The server prints that string on its terminal.

Running the Client in MATLAB

The server must be started before the client. Since Ice for MATLAB does not support server-side behavior, we need to use a server from another language mapping. In this case, we will use the C++ server:

```
server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a MATLAB console:

```
client()
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` message that is produced by the printer. To get rid of the server, we interrupt it on the command line.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Error using Demo.PrinterPrx.checkedCast (line 59)
::Ice::ConnectionRefusedException

Error in Client (line 8)
    printer = Demo.PrinterPrx.checkedCast(base);
```

Note that, to successfully run the client, MATLAB must be able to locate Ice for MATLAB. See the Ice for MATLAB installation instructions for more information.

See Also

- Client-Side Slice-to-MATLAB Mapping
- IceGrid

Writing an Ice Application with Objective-C

This page shows how to create an Ice application with Objective-C.

On this page:

Compiling a Slice Definition for Objective-C

The first step in creating our Objective-C application is to compile our Slice definition to generate Objective-C proxies and skeletons. You can compile the definition as follows:

```
slice2objc Printer.ice
```

The `slice2objc` compiler produces two Objective-C source files from this definition, `Printer.h` and `Printer.m`.

- `Printer.h`
  The `Printer.h` header file contains Objective-C type definitions that correspond to the Slice definitions for our `Printer` interface. This header file must be included in both the client and the server source code.
- `Printer.m`
  The `Printer.m` file contains the source code for our `Printer` interface. The generated source contains type-specific run-time support for both clients and servers. For example, it contains code that marshals parameter data (the string passed to the `printString` operation) on the client side and unmarshals that data on the server side.
  The `Printer.m` file must be compiled and linked into both client and server.

Writing and Compiling a Server in Objective-C

The source code for the server takes only a few lines and is shown in full here:

Objective-C

```
#import <objc/Ice.h>
#import <Printer.h>
#import <stdio.h>

@interface PrinterI : DemoPrinter <DemoPrinter>
@end

@implementation PrinterI
-(void) printString:(NSMutableString *)s current:(ICECurrent *)current
{
    printf("%s\n", [s UTF8String]);
    fflush(stdout);
}
@end

int
main(int argc, char* argv[])
{
    int status = EXIT_FAILURE;
    @autoreleasepool
    {
        id<ICECommunicator> communicator = nil;
        @try
        {
            communicator = [ICEUtil createCommunicator:&argc argv:argv];
            id<ICEObjectAdapter> adapter = [communicator
createObjectAdapterWithEndpoints:@"SimplePrinterAdapter"

endpoints:@"default -p 10000"];
            [adapter add:[PrinterI printer] identity:[ICEUtil
stringToIdentity:@"SimplePrinter"]];
            [adapter activate];

            [communicator waitForShutdown];

            status = EXIT_SUCCESS;
        }
        @catch(NSException* ex)
        {
            NSLog(@"%@", ex);
        }

        [communicator destroy];
    }
    return status;
}
```

Every Ice source file starts with an include directive for `objc/Ice.h`, which contains the definitions for the Ice run time. We also include `Printer.h`, which was generated by the Slice compiler and contains the Objective-C definitions for our printer interface:

Objective-C

```
#import <objc/Ice.h>
#import <Printer.h>
#import <stdio.h>
```

Our server implements a single printer servant, of type `PrinterI`. Looking at the generated code in `Printer.h`, we find the following (tidied up a little to get rid of irrelevant detail):

Objective-C

```
@protocol DemoPrinter <ICEObject>
-(void) printString:(NSMutableString *)s
            current:(ICECurrent *)current;
@end

@interface DemoPrinter : ICEObject
// ...
@end
```

The `DemoPrinter` protocol and class definitions are generated by the Slice compiler. The protocol defines the `printString` method, which we must implement in our servant. The `DemoPrinter` class contains methods that are internal to the mapping, so we are not concerned with these. However, our servant must derive from this skeleton class:

Objective-C

```
@interface PrinterI : DemoPrinter <DemoPrinter>
@end

@implementation PrinterI
-(void) printString:(NSMutableString *)s current:(ICECurrent *)current
{
    printf("%s\n", [s UTF8String]);
    fflush(stdout);
}
@end
```

As you can see, the implementation of the `printString` method is trivial: it simply writes its string argument to `stdout`.

Note that `printString` has a second parameter of type `ICECurrent`. The Ice run time passes additional information about an incoming request to the servant in this parameter. For now, we will ignore it.

What follows is the server main program. Note the general structure of the code:

Objective-C

```
int
main(int argc, char* argv[])
{
    int status = EXIT_FAILURE;
    @autoreleasepool
    {
        id<ICECommunicator> communicator = nil;
        @try
        {
            communicator = [ICEUtil createCommunicator:&argc argv:argv];

            // Server implementation here...

            status = EXIT_SUCCESS;
        }
        @catch(NSException* ex)
        {
            NSLog(@"%@", ex);
        }

        [communicator destroy];
    }
    return status;
}
```

The body of `main` instantiates an autorelease pool, which it releases before returning to ensure that the program does not leak memory. `main` contains the declaration of two variables, `status` and `communicator`. The `status` variable contains the exit status of the program and the `communicator` variable, of type `id<ICECommunicator>`, contains the main handle to the Ice run time.

Following these declarations is a `try` block in which we place all the server code, followed by a `catch` handler that logs any unhandled exceptions.

Before returning, `main` executes a bit of cleanup code that calls the `destroy` method on the communicator. The cleanup call is outside the `try` block for a reason: we must ensure that the Ice run time is finalized whether the code terminates normally or terminates due to an exception.

> Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

The body of the `try` block contains the actual server code:

Objective-C

```
communicator = [ICEUtil createCommunicator:&argc argv:argv];
            id<ICEObjectAdapter> adapter = [communicator
createObjectAdapterWithEndpoints:@"SimplePrinterAdapter"

endpoints:@"default -p 10000"];
            [adapter add:[PrinterI printer] identity:[ICEUtil
stringToIdentity:@"SimplePrinter"]];
            [adapter activate];

            [communicator waitForShutdown];
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `createCommunicator`. (We pass `argc` and `argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `createCommunicator` returns a value of type `id<ICECommunicator>`, which is the main object in the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.) The server starts to process incoming requests from clients as soon as the adapter is activated.
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Assuming that we have the server code in a file called `Server.m`, we can compile it as follows:

```
c++ -c -I. -I$ICE_HOME/include Printer.m Server.m
```

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the `ICE_HOME` environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `/opt/ice`, set `ICE_HOME` to that path.) Depending on your platform, you may have to add additional include directives or other options to the compiler; please see the demo programs that ship with Ice for the details.

Finally, we need to link the server into an executable:

```
c++ Printer.o Server.o -o server -L$ICE_HOME/lib -lIceObjC -framework
Foundation
```

Again, depending on the platform, the actual list of libraries you need to link against may be longer. The demo programs that ship with Ice contain all the details.

Writing and Compiling a Client in Objective-C

The client code looks very similar to the server. Here it is in full:

Objective-C

```
#import <objc/Ice.h>
#import <Printer.h>
#import <stdio.h>

int
main(int argc, char* argv[])
{
    int status = EXIT_FAILURE;
    @autoreleasepool
    {
        id<ICECommunicator> communicator = nil;
        @try
        {
            communicator = [ICEUtil createCommunicator:&argc argv:argv];
            id<ICEObjectPrx> base = [communicator
stringToProxy:@"SimplePrinter:default -p 10000"];
            id<DemoPrinterPrx> printer = [DemoPrinterPrx
checkedCast:base];
            if(!printer)
            {
                [NSException raise:@"Invalid proxy" format:@""];
            }
            [printer printString:@"Hello World!"];
            status = EXIT_SUCCESS;
        }
        @catch(NSException* ex)
        {
            NSLog(@"%@", ex);
        }

        [communicator destroy];
    }
    return status;
}
```

Note that the overall code layout is the same as for the server: we include the headers for the Ice run time and the header generated by the Slice compiler, and we use the same `try` block and `catch` handlers to deal with errors.

The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `createCommunicator`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `id<ICEObjectPrx>`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling the `checkedCast` class method on the `DemoPrinterPrx` class. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy to a `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns a null proxy.

4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling and linking the client looks much the same as for the server:

```
c++ -c -I. -I$ICE_HOME/include Printer.m Client.m
c++ Printer.o Client.o -o client -L$ICE_HOME/lib -lIceObjC -framework
Foundation
```

Running Client and Server in Objective-C

To run client and server, we first start the server in a separate window:

```
./server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
./client
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get:

```
Network.cpp:1218: Ice::ConnectionRefusedException:
connection refused: Connection refused
```

Note that, to successfully run client and server, you may have to set `DYLD_LIBRARY_PATH` to include the Ice library directory. Please see the installation instructions and the demo applications that ship with Ice Touch for details.

See Also

- Client-Side Slice-to-Objective-C Mapping
- Server-Side Slice-to-Objective-C Mapping
- The Current Object
- IceGrid

Writing an Ice Application with PHP

This page shows how to create an Ice client application with PHP.

On this page:

Compiling a Slice Definition for PHP

The first step in creating our PHP application is to compile our Slice definition to generate PHP code. You can compile the definition as follows:

```
slice2php Printer.ice
```

The `slice2php` compiler produces a single source file, `Printer.php`, from this definition. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Client in PHP

The client code, in `Client.php`, is shown below in full:

PHP

```
<?php
require 'Ice.php';
require 'Printer.php';

$ic = null;
try
{
    $ic = Ice\initialize();
    $base = $ic->stringToProxy("SimplePrinter:default -p 10000");
    $printer = Demo\PrinterPrxHelper::checkedCast($base);
    if(!$printer)
    {
        throw new RuntimeException("Invalid proxy");
    }

    $printer->printString("Hello World!");
}
catch(Exception $ex)
{
    echo $ex;
}

if($ic)
{
    $ic->destroy(); // Clean up
}
?>
```

The program begins with `require` statements to load the Ice run-time definitions (`Ice.php`) and the code we generated from our Slice definition in the previous section (`Printer.php`).

The body of the main program contains a `try` block in which we place all the client code, followed by a `catch` block. The `catch` block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to the main program, which prints the exception and then returns failure to the operating system.

The body of our `try` block goes through the following steps:

1. We initialize the Ice run time by calling `Ice\initialize`. The call to `initialize` returns an `Ice\Communicator` reference, which is the main object in the Ice run time.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice\ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo\PrinterPrxHelper::checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy narrowed to the `Printer` interface ; otherwise, if the proxy denotes an interface of some other type, the call returns `null`.
4. We test that the down-cast succeeded and, if not, throw an exception that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the

Ice run time. If a script neglects to destroy the communicator, Ice destroys it automatically.

Running the Client in PHP

The server must be started before the client. Since Ice for PHP does not support server-side behavior, we need to use a server from another language mapping. In this case, we will use the C++ server:

```
server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window using PHP's command-line interpreter:

```
php -f Client.php
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
exception ::Ice::ConnectionRefusedException
{
    error = 111
}
```

Note that, to successfully run the client, the PHP interpreter must be able to locate the Ice extension for PHP. See the Ice for PHP installation instructions for more information.

See Also

- Client-Side Slice-to-PHP Mapping
- IceGrid

Writing an Ice Application with Python

This page shows how to create an Ice application with Python.

On this page:

Compiling a Slice Definition for Python

The first step in creating our Python application is to compile our Slice definition to generate Python proxies and skeletons. You can compile the definition as follows:

```
slice2py Printer.ice
```

The `slice2py` compiler produces a single source file, `Printer_ice.py`, from this definition. The compiler also creates a Python package for the `Demo` module, resulting in a subdirectory named `Demo`. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Server in Python

To implement our `Printer` interface, we must create a servant class. By convention, a servant class uses the name of its interface with an `I`-suffix, so our servant class is called `PrinterI`:

Python

```
class PrinterI(Demo.Printer):
    def printString(self, s, current=None):
        print s
```

The `PrinterI` class inherits from a base class called `Demo.Printer`, which is generated by the `slice2py` compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code, in `server.py`, follows our servant class and is shown in full here:

Python

```
import sys, Ice
import Demo

class PrinterI(Demo.Printer):
    def printString(self, s, current=None):
        print s

with Ice.initialize(sys.argv) as communicator:
    adapter = communicator.createObjectAdapterWithEndpoints("SimplePrin
terAdapter", "default -p 10000")
    object = PrinterI()
    adapter.add(object, communicator.stringToIdentity("SimplePrinter"))
    adapter.activate()
    communicator.waitForShutdown()
```

The body of the main program contains a `with` block in which we place all the server code. If the code throws an exception, it will be handled by the Python interpreter which typically prints out the exception and then returns failure to the operating system.

The `Ice.Communicator` object implements the Python context manager protocol, which allows us to use the `with` statement for the initialization of the `Ice.Communicator` object. This ensures the communicator `destroy` method is called when the `with` block goes out of scope. Doing this is essential in order to correctly finalize the Ice run time

> Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

The server code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.initialize`. (We pass `sys.argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Writing a Client in Python

The client code, in `client.py`, looks very similar to the server. Here it is in full:

Python

```
import sys, Ice
import Demo

with Ice.initialize(sys.argv) as communicator:
    base = communicator.stringToProxy("SimplePrinter:default -p 10000")
    printer = Demo.PrinterPrx.checkedCast(base)
    if not printer:
        raise RuntimeError("Invalid proxy")

    printer.printString("Hello World!")
```

Note that the overall code layout is the same as for the server: we use the same `with` block. The code does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
3. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo.PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo.PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `None`.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Running Client and Server in Python

To run client and server, we first start the server in a separate window:

```
python server.py
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
python client.py
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Traceback (most recent call last):
  File "client.py", line 10, in ?
    printer = Demo.PrinterPrx.checkedCast(base)
  File "Printer_ice.py", line 43, in checkedCast
    return Demo.PrinterPrx.ice_checkedCast(proxy, '::Demo::Printer', fa
cet)
ConnectionRefusedException: Ice.ConnectionRefusedException:
Connection refused
```

Note that, to successfully run the client and server, the Python interpreter must be able to locate the Ice extension for Python. See the Ice for Python installation instructions for more information.

See Also

- Client-Side Slice-to-Python Mapping
- Server-Side Slice-to-Python Mapping
- The Current Object
- IceGrid

Writing an Ice Application with Ruby

This page shows how to create an Ice client application with Ruby.

On this page:

Compiling a Slice Definition for Ruby

The first step in creating our Ruby application is to compile our Slice definition to generate Ruby proxies. You can compile the definition as follows:

```
slice2rb Printer.ice
```

The `slice2rb` compiler produces a single source file, `Printer.rb`, from this definition. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Client in Ruby

The client code, in `client.rb`, is shown below in full:

Ruby

```
require 'Printer.rb'

Ice::initialize(ARGV) do |communicator|
    base = communicator.stringToProxy("SimplePrinter:default -h
localhost -p 10000")
    printer = Demo::PrinterPrx::checkedCast(base)
    if not printer
        raise "Invalid proxy"
    end

    printer.printString("Hello World!")
end
```

The program begins with a `require` statement, which loads the Ruby code we generated from our Slice definition in the previous section. It is not necessary for the client to explicitly load the `Ice` module because `Printer.rb` does that for you.

The body of the main program goes through the following steps:

1. We initialize the Ice run time by calling `Ice::initialize`. (We pass ARGV to this call because the client may have command-line arguments that are of interest to the run time; for this example, the client does not require any command-line arguments.)
2. We do our work inside a block. The block accepts an `Ice::Communicator` reference, which is the main object in the Ice run time. `Ice::initialize` will automatically destroy the communicator when the block completes.
3. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss IceGrid.)
4. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo::PrinterPrx::checkedCast`. A checked cast sends a message to the server,

effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo::PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `nil`.
5. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
6. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Running the Client in Ruby

The server must be started before the client. Since Ice for Ruby does not support server-side behavior, we need to use a server from another language mapping. In this case, we will use the C++ server:

```
server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
ruby client.rb
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
exception ::Ice::ConnectionRefusedException
{
    error = 111
}
```

Note that, to successfully run the client, the Ruby interpreter must be able to locate the Ice extension for Ruby. See the Ice for Ruby installation instructions for more information.

See Also

- Client-Side Slice-to-Ruby Mapping
- IceGrid

The Slice Language

Slice (Specification Language for Ice) is the fundamental abstraction mechanism for separating object interfaces from their implementations. Slice establishes a contract between client and server that describes the interfaces, operations and parameter types used by an application. This description is independent of the implementation language, so it does not matter whether the client is written in the same language as the server.

> Even though Slice is an acronym, it is pronounced as a single syllable, like a slice of bread.

Slice definitions are compiled for a particular implementation language by a compiler. The language-specific Slice compiler translates the language-independent Slice definitions into language-specific type definitions and APIs. These types and APIs are used by the developer to provide application functionality and to interact with Ice. The translation algorithms for various implementation languages are known as *language mappings*, and Ice provides a number of language mappings (for C++, C#, Java, JavaScript, Python and more).

Because Slice describes interfaces and types (but not implementations), it is a purely declarative language; there is no way to write executable statements in Slice.

Slice definitions focus on object interfaces, the operations supported by those interfaces, and exceptions that may be raised by operations. This requires quite a bit of supporting machinery; in particular, much of Slice is concerned with the definition of data types. This is because data can be exchanged between client and server only if their types are defined in Slice. You cannot exchange arbitrary C++ data between a client and a server because it would destroy the language independence of Ice. However, you can always create a Slice type definition that corresponds to the C++ data you want to send, and then you can transmit the Slice type.

We present the full syntax and semantics of Slice here. Because much of Slice is based on C++ and Java, we focus on those areas where Slice differs from C++ or Java or constrains the equivalent C++ or Java feature in some way. Slice features that are identical to C++ and Java are mentioned mostly by example.

Topics

Slice Compilation

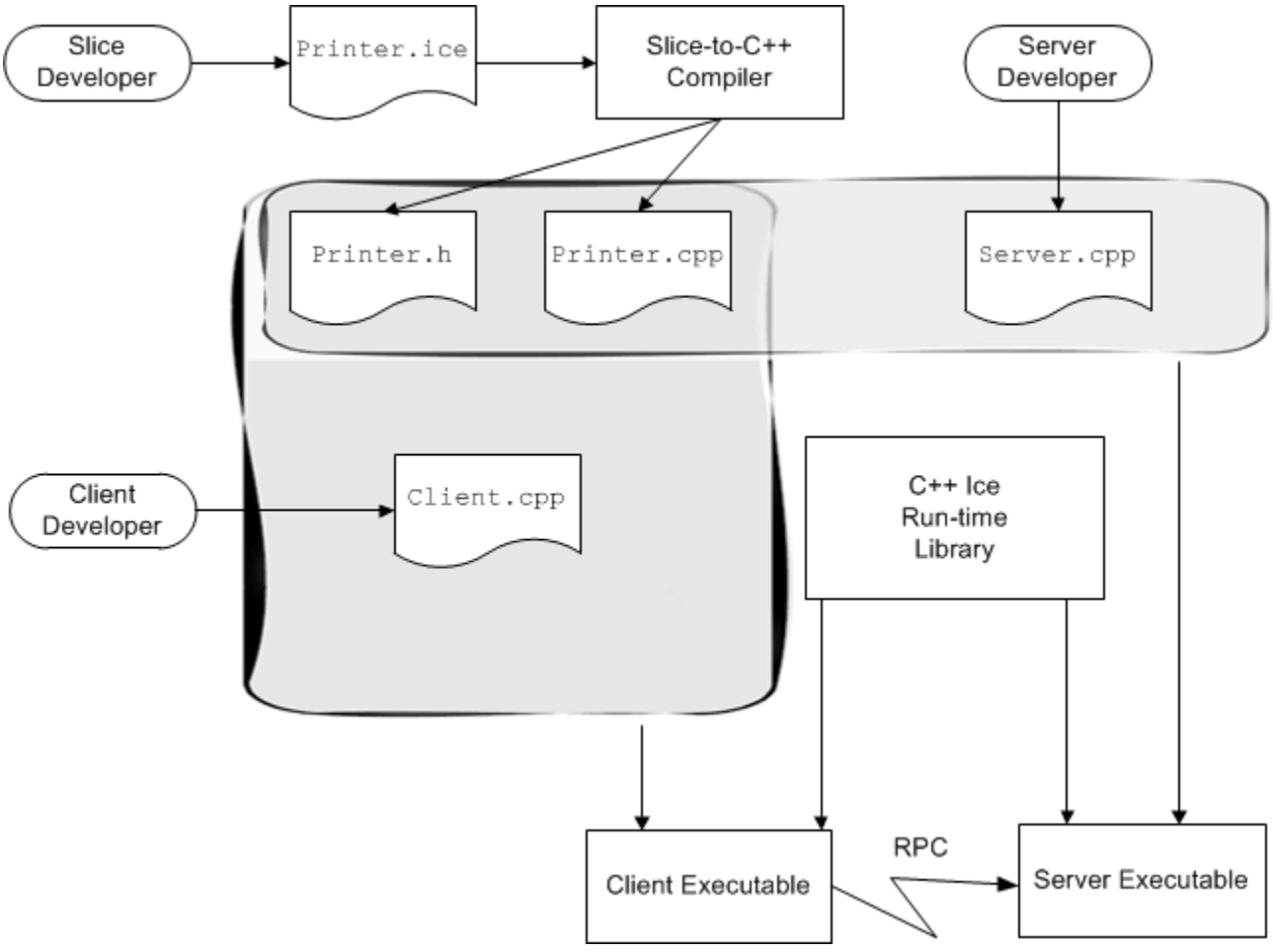
On this page:

Compilation

A Slice compiler produces source files that must be combined with application code to produce client and server executables.

Single Development Environment for Client and Server

The figure below shows the situation when both client and server are developed in C++. The Slice compiler generates two files from a Slice definition in a source file `Printer.ice`: a header file (`Printer.h`) and a source file (`Printer.cpp`)



Development process if client and server share the same development environment.

- The `Printer.h` header file contains definitions that correspond to the types used in the Slice definition. It is included in the source code of both client and server to ensure that client and server agree about the types and interfaces used by the application.
- The `Printer.cpp` source file provides an API to the client for sending messages to remote objects. The client source code (`Client.cpp`, written by the client developer) contains the client-side application logic. The generated source code and the client code are compiled and linked into the client executable.

The `Printer.cpp` source file also contains source code that provides an up-call interface from the Ice run time into the server code written

by the developer and provides the connection between the networking layer of Ice and the application code. The server implementation file (`Server.cpp`, written by the server developer) contains the server-side application logic (the object implementations, properly termed *servants*). The generated source code and the implementation source code are compiled and linked into the server executable.

Both client and server also link with an Ice library that provides the necessary run-time support.

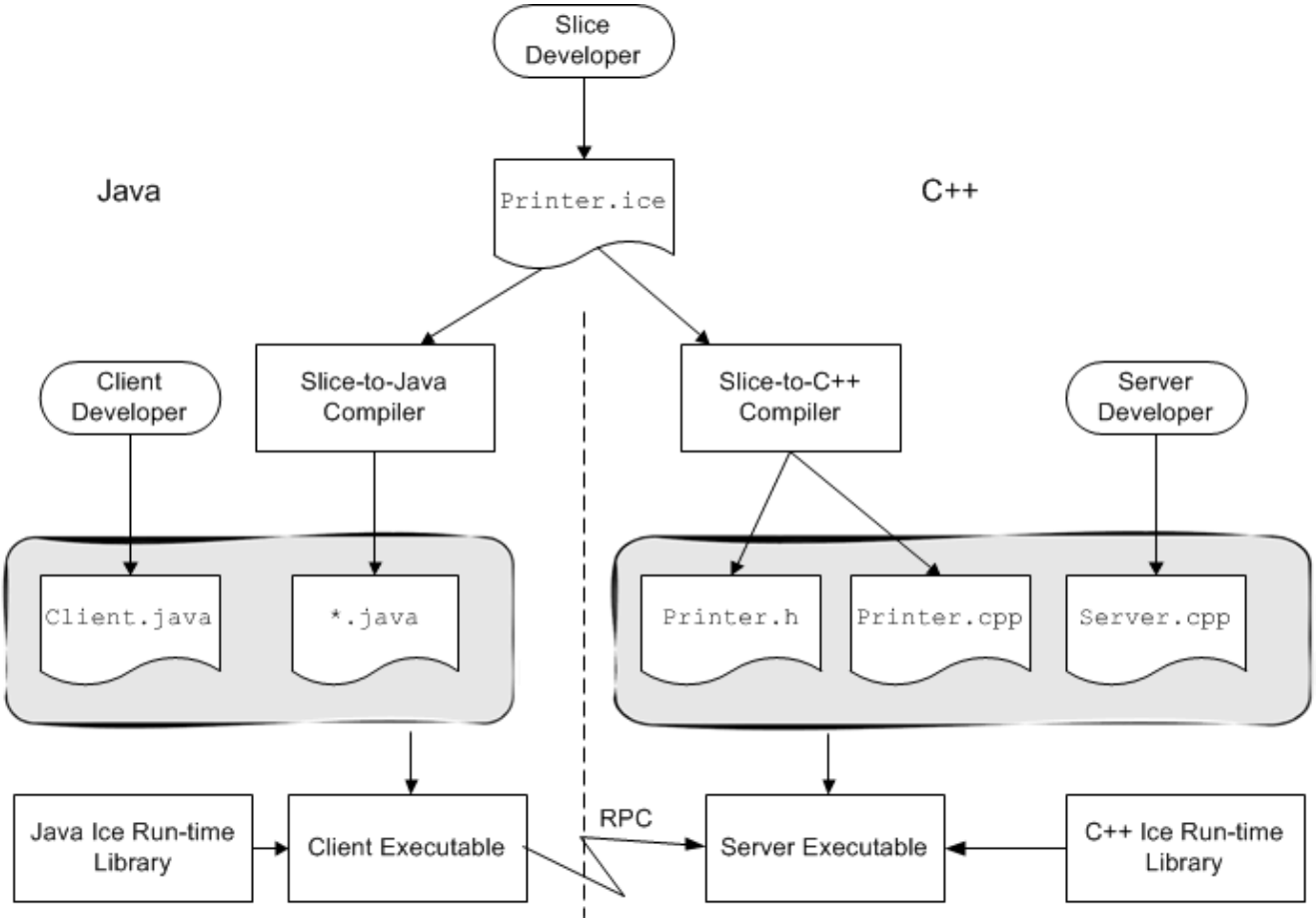
You are not limited to a single implementation of a client or server. For example, you can build multiple servers, each of which implements the same interfaces but uses different implementations (for example, with different performance characteristics). Multiple such server implementations can coexist in the same system. This arrangement provides one fundamental scalability mechanism in Ice: if you find that a server process starts to bog down as the number of objects increases, you can run an additional server for the same interfaces on a different machine. Such *federated* servers provide a single logical service that is distributed over a number of processes on different machines. Each server in the federation implements the same interfaces but hosts different object instances. (Of course, federated servers must somehow ensure consistency of any databases they share across the federation.)

Ice also provides support for *replicated* servers. Replication permits multiple servers to each implement the same set of object instances. This improves performance and scalability (because client load can be shared over a number of servers) as well as redundancy (because each object is implemented in more than one server).

Different Development Environments for Client and Server

Client and server cannot share any source or binary components if they are developed in different languages. For example, a client written in Java cannot include a C++ header file.

This figure shows the situation when a client written in Java and the corresponding server is written in C++. In this case, the client and server developers are completely independent, and each uses his or her own development environment and language mapping. The only link between client and server developers is the Slice definition each one uses.



Development process for different development environments.

For Java, the slice compiler creates a number of files whose names depend on the names of various Slice constructs. (These files are collectively referred to as `*.java` in the above figure.)

Slice Compilation and your Build Environment

One way to integrate Slice compilation in your build system is to compile your Slice files manually, and then keep (check-in) the generated files like other source files. Later on, each time you change a Slice file, you have to remember to recompile this Slice file and update the generated files. While simple, this approach can lead to inconsistencies and bugs if you forget to recompile a modified Slice file.

We recommend you use instead an *Ice Builder* for your build environment to manage the compilation of your Slice files. An Ice Builder is a simple plug-in or task for your build environment that compiles or recompiles Slice files when it detects the corresponding generated files are missing or out of date. A Builder performs this Slice compilation by invoking the Slice compiler for the target programming language–it does compile the files itself and usually supports several versions of Ice.

See Also

- Using the Slice Compilers
- Ice Builders

Slice Source Files

Slice defines a number of rules for the naming and contents of Slice source files.

On this page:

File Naming

Files containing Slice definitions must end in a `.ice` file extension, for example, `Clock.ice` is a valid file name. Other file extensions are rejected by the compilers.

For case-insensitive file systems, the file extension may be written as uppercase or lowercase, so `Clock.ICE` is legal. For case-sensitive file systems (such as Unix), `Clock.ICE` is illegal. (The extension must be in lowercase.)

File Format

Slice is a free-form language so you can use spaces, horizontal and vertical tab stops, form feeds, and newline characters to lay out your code in any way you wish. (White space characters are token separators). Slice does not attach semantics to the layout of a definition. You may wish to follow the style we have used for the Slice examples throughout this book.

Slice files can be ASCII text files or use the UTF-8 character encoding with an optional byte order marker (BOM) at the beginning of each file. However, Slice identifiers are limited to ASCII letters and digits; non-ASCII letters can appear only in comments and string literals.

Preprocessing

Slice supports the same preprocessor directives as C++, so you can use directives such as `#include` and macro definitions. However, Slice permits `#include` directives only at the beginning of a file, before any Slice definitions.

If you use `#include` directives, it is a good idea to protect them with guards to prevent double inclusion of a file:

Slice

```
// File Clock.ice
#ifndef _CLOCK_ICE
#define _CLOCK_ICE

// #include directives here...
// Definitions here...

#endif _CLOCK_ICE
```

The following `#pragma` directive offers a simpler way to achieve the same result:

Slice

```
// File Clock.ice
#pragma once

// #include directives here...
// Definitions here...
```

`#include` directives permit a Slice definition to use types defined in a different source file. The Slice compilers parse all of the code in a source file, including the code in subordinate `#include` files. However, the compilers generate code only for the top-level file(s) nominated on the command line. You must separately compile subordinate `#include` files to obtain generated code for all the files that make up your Slice definition.

Note that you should avoid `#include` with double quotes:

Slice

```
#include "Clock.ice" // Not recommended!
```

While double quotes will work, the directory in which the preprocessor tries to locate the file can vary depending on the operating system, so the included file may not always be found where you expect it. Instead, use angle brackets (<>); you can control which directories are searched for the file with the `-I` option of the Slice compiler.

Also note that, if you include a path separator in a `#include` directive, you must use a forward slash:

Slice

```
#include <SliceDefs/Clock.ice>  // OK
```

You cannot use a backslash in `#include` directives:

Slice

```
#include <SliceDefs\Clock.ice>  // Illegal
```

Detecting Ice Versions

The Slice compilers define the preprocessor macro `__ICE_VERSION__` with a numeric representation of the Ice version. The value of this macro is the same as the C++ macro `ICE_INT_VERSION`. You can use this macro to make your Slice definitions backward-compatible with older Ice releases, while still taking advantage of newer Ice features when possible. For example, the Slice definition shown below makes use of custom enumerator values:

Slice

```
#if defined(__ICE_VERSION__) && __ICE_VERSION__ >= 030500
enum Fruit { Apple, Pear = 3, Orange }
#else
enum Fruit { Apple, Pear, Orange }
#endif
```

Although this example is intended to show how to use the `ICE_VERSION` macro, it also highlights a potential pitfall that you must be aware of when trying to maintain backward compatibility: the two definitions of `Fruit` are not wire-compatible.

Detecting Slice Compilers

Each Slice compiler defines its own macro so that you can customize your Slice code for certain language mappings. The following macros are defined by their respective compilers:

```
__SLICE2JAVA__
__SLICE2JS__
__SLICE2CPP__
__SLICE2CS__
__SLICE2PY__
__SLICE2PHP__
__SLICE2RB__
__SLICE2FREEZE__
__SLICE2FREEZEJ__
__SLICE2MATLAB__
__TRANSFORMDB__
__DUMPDB__
```

For example, .NET developers may elect to avoid the use of default values for structure members because the presence of default values changes the C# mapping of the structure from `struct` to `class`:

Slice

```
struct Record
{
    // ...
#if __SLICE2CS__
    bool active;
#else
    bool active = true;
#endif
}
```

Definition Order

Slice constructs, such as modules, interfaces, or type definitions, can appear in any order you prefer. However, identifiers must be declared before they can be used.

See Also

- Using the Slice Compilers

Lexical Rules

Slice's lexical rules are very similar to those of C++ and Java, except for some differences for identifiers.

On this page:

Comments

Slice definitions permit both the C and the C++ style of writing comments:

Slice

```
/*
 * C-style comment.
 */

// C++-style comment extending to the end of this line.
```

Keywords

Slice uses a number of keywords, which must be spelled in lowercase. For example, `class` and `dictionary` are keywords and must be spelled as shown. There are three exceptions to this lowercase rule: `LocalObject`, `Object` and `Value` are keywords and must be capitalized as shown.

Identifiers

Identifiers begin with an alphabetic character followed by any number of alphabetic characters or digits. Underscores are also permitted in identifiers with the following limitations:

- an identifier cannot begin or end with an underscore
- an identifier cannot contain multiple consecutive underscores

Given these rules, the identifier `get_account_name` is legal but not `_account`, `account_`, or `get__account`.

Slice identifiers are restricted to the ASCII range of alphabetic characters and cannot contain non-English letters, such as Å. (Supporting non-ASCII identifiers would make it very difficult to map Slice to target languages that lack support for this feature.)

Case Sensitivity

Identifiers are case-insensitive but must be capitalized consistently. For example, `TimeOfDay` and `TIMEOFDAY` are considered the same identifier within a naming scope. However, Slice enforces consistent capitalization. After you have introduced an identifier, you must capitalize it consistently throughout; otherwise, the compiler will reject it as illegal. This rule exists to permit mappings of Slice to languages that ignore case in identifiers as well as to languages that treat differently capitalized identifiers as distinct.

Identifiers That Are Keywords

You can define Slice identifiers that are keywords in one or more implementation languages. For example, `switch` is a perfectly good Slice identifier but is a C++ and Java keyword. Each language mapping defines rules for dealing with such identifiers. The solution typically

involves using a prefix to map away from the keyword. For example, the Slice identifier `switch` is mapped to `_cpp_switch` in C++ and `_switch` in Java.

The rules for dealing with keywords can result in hard-to-read source code. Identifiers such as `native`, `throw`, or `export` will clash with C++ or Java keywords (or both). To make life easier for yourself and others, try to avoid Slice identifiers that are implementation language keywords. Keep in mind that mappings for new languages may be added to Ice in the future. While it is not reasonable to expect you to compile a list of all keywords in all popular programming languages, you should make an attempt to avoid at least common keywords. Slice identifiers such as `self`, `import`, and `while` are definitely not a good idea.

Escaped Identifiers

It is possible to use a Slice keyword as an identifier by prefixing the keyword with a backslash, for example:

Slice

```
struct dictionary     // Error!
{
    // ...
}

struct \dictionary    // OK
{
    // ...
}

struct \foo           // Legal, same as "struct foo"
{
    // ...
}
```

The backslash escapes the usual meaning of a keyword; in the preceding example, `\dictionary` is treated as the identifier `dictionary`. The escape mechanism exists to permit keywords to be added to the Slice language over time with minimal disruption to existing specifications: if a pre-existing specification happens to use a newly-introduced keyword, that specification can be fixed by simply prepending a backslash to the new keyword. Note that, as a matter of style, you should avoid using Slice keywords as identifiers (even though the backslash escapes allow you to do this).

It is legal (though redundant) to precede an identifier that is not a keyword with a backslash — the backslash is ignored in that case.

Reserved Identifiers

Slice reserves the identifier `Ice` and all identifiers beginning with `Ice` (in any capitalization) for the Ice implementation. For example, if you try to define a type named `Icecream`, the Slice compiler will issue an error message.

> You can suppress this behavior by using the `ice-prefix` Slice metadata directive, which enables definition of identifiers beginning with `Ice`. However, do not use this directive unless you are compiling the Slice definitions for the Ice run time itself.

Slice identifiers ending in any of the suffixes `Async`, `Disp`, `Helper`, `Holder`, `Prx`, and `Ptr` are also reserved. These endings are used by the various language mappings and are reserved to prevent name clashes in the generated code.

See Also

- Slice Keywords

Modules

On this page:

Modules Reduce Clutter

A common problem in large systems is pollution of the global namespace: over time, as isolated systems are integrated, name clashes become quite likely. Slice provides the `module` construct to alleviate this problem:

Slice

```
module ZeroC
{
    module Client
    {
        // Definitions here...
    }
    module Server
    {
        // Definitions here...
    }
}
```

A module can contain any legal Slice construct, including other module definitions. Using modules to group related definitions together avoids polluting the global namespace and makes accidental name clashes quite unlikely. (You can use a well-known name, such as a company or product name, as the name of the outermost module.)

Modules are Mandatory

Slice requires all definitions to be nested inside a module, that is, you cannot define anything other than a module at global scope. For example, the following is illegal:

Slice

```
interface I   // Error: only modules can appear at global scope
{
    // ...
}
```

Definitions at global scope are prohibited because they cause problems with some implementation languages (such as Python, which does not have a true global scope).

> Throughout the Ice manual, you will occasionally see Slice definitions that are not nested inside a module. This is to keep the examples short and free of clutter. Whenever you see such a definition, assume that it is nested in module `M`.

Reopening Modules

Modules can be reopened:

Slice

```
module ZeroC
{
    // Definitions here...
}

// Possibly in a different source file:

module ZeroC // OK, reopened module
{
    // More definitions here...
}
```

Reopened modules are useful for larger projects: they allow you to split the contents of a module over several different source files. The advantage of doing this is that, when a developer makes a change to one part of the module, only files dependent on the changed part need be recompiled (instead of having to recompile all files that use the module).

Module Mapping

Modules map to a corresponding scoping construct in each programming language. (For example, for C++ and C#, Slice modules map to namespaces whereas, for Java, they map to packages.) This allows you to use an appropriate C++ `using` or Java `import` declaration to avoid excessively long identifiers in your source code.

The Ice Module

APIs for the Ice run time, apart from a small number of language-specific calls that cannot be expressed in Ice, are defined in the `Ice` module. In other words, most of the Ice API is actually expressed as Slice definitions. The advantage of doing this is that a single Slice definition is sufficient to define the API for the Ice run time for all supported languages. The respective language mapping rules then determine the exact shape of each Ice API for each implementation language.

We will incrementally explore the contents of the `Ice` module throughout this manual.

See Also

- Slice Source Files

Basic Types

On this page:

Built-In Basic Types

Slice provides a number of built-in basic types, as shown in this table:

Type	Range of Mapped Type	Size of Mapped Type
`bool`	`false` or `true`	1bit
`byte`	-128-127 or 0-255 [a]	8 bits
`short`	-2^{15} to $2^{15}-1$	16 bits
`int`	-2^{31} to $2^{31}-1$	32 bits
`long`	-2^{63} to $2^{63}-1$	64 bits
`float`	IEEE single-precision	32 bits
`double`	IEEE double-precision	64 bits
`string`	All Unicode characters, excluding the character with all bits zero.	Variable-length

[a] The range depends on whether `byte` maps to a signed or an unsigned type.

All the basic types (except `byte`) are subject to changes in representation as they are transmitted between clients and servers. For example, a `long` value is byte-swapped when sent from a little-endian to a big-endian machine. Similarly, strings undergo translation in representation if they are sent, for example, from an EBCDIC to an ASCII implementation, and the characters of a string may also change in size. (Not all architectures use 8-bit characters). However, these changes are transparent to the programmer and do exactly what is required.

Integer Types

Slice provides integer types `short`, `int`, and `long`, with 16-bit, 32-bit, and 64-bit ranges, respectively. Note that, on some architectures, any of these types may be mapped to a native type that is wider. Also note that no unsigned types are provided. (This choice was made because unsigned types are difficult to map into languages without native unsigned types, such as Java. In addition, the unsigned integers add little value to a language. (See [1] for a good treatment of the topic.)

Floating-Point Types

These types follow the IEEE specification for single- and double-precision floating-point representation [2]. If an implementation cannot support IEEE format floating-point values, the Ice run time converts values into the native floating-point representation (possibly at a loss of precision or even magnitude, depending on the capabilities of the native floating-point format).

Strings

Slice strings use the Unicode character set. The only character that cannot appear inside a string is the zero character.

> This decision was made as a concession to C++, with which it becomes impossibly difficult to manipulate strings with embedded zero characters using standard library routines, such as `strlen` or `strcat`.

The Slice data model does not have the concept of a null string (in the sense of a C++ null pointer). This decision was made because null strings are difficult to map to languages without direct support for this concept (such as Python). Do not design interfaces that depend on a

null string to indicate "not there" semantics. If you need the notion of an optional string, use a class, a sequence of strings, or use an empty string to represent the idea of a null string. (Of course, the latter assumes that the empty string is not otherwise used as a legitimate string value by your application.)

Booleans

Boolean values can have only the values `false` and `true`. Language mappings use the corresponding native boolean type if one is available.

Bytes

The Slice type `byte` is an (at least) 8-bit type that is guaranteed not to undergo any changes in representation as it is transmitted between address spaces. This guarantee permits exchange of binary data such that it is not tampered with in transit. All other Slice types are subject to changes in representation during transmission.

See Also

- Sequences
- Classes

References

1. Lakos, J. 1996. Large-Scale C++ Software Design. Reading, MA: Addison-Wesley.
2. Institute of Electrical and Electronics Engineers. 1985. *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*. Piscataway, NJ: Institute of Electrical and Electronic Engineers.

User-Defined Types

In addition to providing the built-in basic types, Slice allows you to define complex types: enumerations, structures, sequences, and dictionaries.

Topics

Enumerations

Enumeration Syntax and Semantics

A Slice enumerated type definition looks identical to C++:

Slice

```
module M
{
    enum Fruit { Apple, Pear, Orange }
}
```

This definition introduces a type named `Fruit` that becomes a new type in its own right. Slice guarantees that the values of enumerators increase from left to right, so `Apple` compares less than `Pear` in every language mapping. By default, the first enumerator has a value of zero, with sequentially increasing values for subsequent enumerators.

A Slice enum type introduces a new namespace scope, so the following is legal:

Slice

```
module M
{
    enum Fruit { Apple, Pear, Orange }
    enum ComputerBrands { Apple, Dell, HP, Lenovo }
}
```

The example below shows how to refer to an enumerator from a different scope:

Slice

```
module M
{
    enum Color { Red, Green, Blue }
}
module N
{
    struct Pixel
    {
        M::Color c = Blue;
    }
}
```

Slice does not permit empty enumerations.

> In Ice releases prior to Ice 3.7, an enum type did not create a new namespace and its enumerators were in the same namespace as the enum type itself. With these releases, you had to select longer enumerator names to avoid a naming clash.

Custom Enumerator Values

Slice also permits you to assign custom values to enumerators:

Slice

```
const int PearValue = 7;
enum Fruit { Apple = 0, Pear = PearValue, Orange }
```

Custom values must be unique and non-negative, and may refer to Slice constants of integer types. If no custom value is specified for an enumerator, its value is one greater than the enumerator that immediately precedes it. In the example above, `Orange` has the value 8.

The maximum value for an enumerator value is the same as the maximum value for `int`, $2^{31} - 1$.

Slice does not require custom enumerator values to be declared in increasing order:

Slice

```
enum Fruit { Apple = 5, Pear = 3, Orange = 1 }   // Legal
```

Note however that when there is an inconsistency between the declaration order and the numerical order of the enumerators, the behavior of comparison operations may vary between language mappings.

> For an application that is still using version 1.0 of the Ice encoding, changing the definition of an enumerated type may break backward compatibility with existing applications. For more information, please refer to the encoding rules for enumerators.

See Also

- Structures
- Sequences
- Dictionaries
- Constants and Literals

Structures

Slice supports structures containing one or more named members of arbitrary type, including user-defined complex types. For example:

Slice

```
module M
{
    struct TimeOfDay
    {
        short hour;         // 0 - 23
        short minute;       // 0 - 59
        short second;       // 0 - 59
    }
}
```

As in C++, this definition introduces a new type called `TimeOfDay`. Structure definitions form a namespace, so the names of the structure members need to be unique only within their enclosing structure.

Data member definitions using a named type are the only construct that can appear inside a structure. It is impossible to, for example, define a structure inside a structure:

Slice

```
struct TwoPoints
{
    struct Point      // Illegal!
    {
        short x;
        short y;
    }
    Point coord1;
    Point coord2;
}
```

This rule applies to Slice in general: type definitions cannot be nested (except for modules, which do support nesting). The reason for this rule is that nested type definitions can be difficult to implement for some target languages and, even if implementable, greatly complicate the scope resolution rules. For a specification language, such as Slice, nested type definitions are unnecessary – you can always write the above definitions as follows (which is stylistically cleaner as well):

Slice

```
struct Point
{
    short x;
    short y;
}

struct TwoPoints      // Legal (and cleaner!)
{
    Point coord1;
    Point coord2;
}
```

You can specify a default value for a data member that has one of the following types:

- An integral type (`byte`, `short`, `int`, `long`)
- A floating point type (`float` or `double`)
- `string`
- `bool`
- `enum`

For example:

Slice

```
struct Location
{
    string name;
    Point pt;
    bool display = true;
    string source = "GPS";
}
```

The legal syntax for literal values is the same as for Slice constants, and you may also use a constant as a default value. The language mapping guarantees that data members are initialized to their declared default values using a language-specific mechanism.

See Also

- Modules
- Basic Types
- Enumerations
- Sequences
- Dictionaries
- Constants and Literals

Sequences

Sequences are variable-length collections of elements:

Slice

```
module M
{
    sequence<Fruit> FruitPlatter;
}
```

A sequence can be empty—that is, it can contain no elements, or it can hold any number of elements up to the memory limits of your platform.

Sequences can contain elements that are themselves sequences. This arrangement allows you to create lists of lists:

Slice

```
module M
{
    sequence<FruitPlatter> FruitBanquet;
}
```

Sequences are used to model a variety of collections, such as vectors, lists, queues, sets, bags, or trees. (It is up to the application to decide whether or not order is important; by discarding order, a sequence serves as a set or bag.)

See Also

- Enumerations
- Structures
- Dictionaries
- Constants and Literals
- Classes

Dictionaries

On this page:

Dictionary Syntax and Semantics

A dictionary is a mapping from a key type to a value type.

For example:

Slice

```
module M
{
    struct Employee
    {
        long   number;
        string firstName;
        string lastName;
    }

    dictionary<long, Employee> EmployeeMap;
}
```

This definition creates a dictionary named `EmployeeMap` that maps from an employee number to a structure containing the details for an employee. Whether or not the key type (the employee number, of type `long` in this example) is also part of the value type (the `Employee` structure in this example) is up to you — as far as Slice is concerned, there is no need to include the key as part of the value.

Dictionaries can be used to implement sparse arrays, or any lookup data structure with non-integral key type. Even though a sequence of structures containing key-value pairs could be used to model the same thing, a dictionary is more appropriate:

- A dictionary clearly signals the intent of the designer, namely, to provide a mapping from a domain of values to a range of values. (A sequence of structures of key-value pairs does not signal that same intent as clearly.)
- At the programming language level, sequences are implemented as vectors (or possibly lists), that is, they are not well suited to model sparsely populated domains and require a linear search to locate an element with a particular value. On the other hand, dictionaries are implemented as a data structure (typically a hash table or red-black tree) that supports efficient searching in $O(\log n)$ average time or better.

Allowable Types for Dictionary Keys and Values

The key type of a dictionary need not be an integral type. For example, we could use the following definition to translate the names of the days of the week:

Slice

```
dictionary<string, string> WeekdaysEnglishToGerman;
```

The server implementation would take care of initializing this map with the key-value pairs `Monday-Montag`, `Tuesday-Dienstag`, and so on.

The value type of a dictionary can be any Slice type. However, the key type of a dictionary is limited to one of the following types:

- Integral types (`short`, `int`, `long`)
- bool
- byte
- `string`
- `enum`
- Structures containing only data members of legal key types

Other complex types, such as dictionaries, and floating-point types (`float` and `double`) cannot be used as the key type. Complex types are disallowed because they complicate the language mappings for dictionaries, and floating-point types are disallowed because representational changes of values as they cross machine boundaries can lead to ill-defined semantics for equality.

See Also

- Basic Types
- Enumerations
- Structures
- Sequences
- Constants and Literals

Constants and Literals

On this page:

Allowable Types for Constants

Slice allows you to define constants for the following types:

- An integral type (`bool`, `byte`, `short`, `int`, `long`)
- A floating point type (`float` or `double`)
- `string`
- `enum`

Here are a few examples:

Slice

```
module M
{
    const bool      AppendByDefault = true;
    const byte      LowerNibble = 0x0f;
    const string    Advice = "Don't Panic!";
    const short     TheAnswer = 42;
    const double    PI = 3.1416;

    enum Fruit { Apple, Pear, Orange }
    const Fruit     FavoriteFruit = Pear;
}
```

The syntax for literals is the same as for C++ and Java (with a few minor exceptions).

Boolean constants

Boolean constants can only be initialized with the keywords `false` and `true`. (You cannot use `0` and `1` to represent `false` and `true`.)

Integer literals

Integer literals can be specified in decimal, octal, or hexadecimal notation.

For example:

Slice

```
const byte TheAnswer = 42;
const byte TheAnswerInOctal = 052;
const byte TheAnswerInHex = 0x2A;       // or 0x2a
```

Be aware that, if you interpret `byte` as a number instead of a bit pattern, you may get different results in different languages. For example, for C++, `byte` maps to `unsigned char` whereas, for Java, `byte` maps to `byte`, which is a signed type.

Note that suffixes to indicate long and unsigned constants (`l`, `L`, `u`, `U`, used by C++) are illegal:

Slice

```
const long Wrong = 0u;          // Syntax error
const long WrongToo = 1000000L; // Syntax error
```

The value of an integer literal must be within the range of its constant type, as shown in the Built-In Basic Types table; otherwise the compiler will issue a diagnostic.

Floating-point literals

Floating-point literals use C++ syntax, except that you cannot use an `l` or `L` suffix to indicate an extended floating-point constant; however, `f` and `F` are legal (but are ignored).

Here are a few examples:

Slice

```
const float P1 = -3.14f;    // Integer & fraction, with suffix
const float P2 = +3.1e-3;   // Integer, fraction, and exponent
const float P3 = .1;        // Fraction part only
const float P4 = 1.;        // Integer part only
const float P5 = .9E5;      // Fraction part and exponent
const float P6 = 5e2;       // Integer part and exponent
```

Floating-point literals must be within the range of the constant type (`float` or `double`); otherwise, the compiler will issue a diagnostic.

String literals

Slice string literals support the same escape sequences as C++, with the exception of hexadecimal escape sequences that are limited to two hexadecimal digits.

Escape Sequence	Name	Corresponding ASCII or Unicode Code Point	Notes
`\'`	single quote	0x27	
`\"`	double quote	0x22	
`\?`	question mark	0x3f	
`\\`	backslash	0x5c	
`\a`	audible bell	0x07	
`\b`	backspace	0x08	
`\f`	form feed	0x0c	
`\n`	line feed	0x0a	
`\r`	carriage return	0x0d	
`\t`	horizontal tab	0x09	
`\v`	vertical tab	0x0b	

`\nnn`	octal escape sequence		1 to 3 octal digits (0-7) that represent a byte value between 0 and 255
`\xnn`	hexadecimal escape sequence		1 to 2 hexadecimal digits (0-9, a-f, A-F)
`\unnnn`	universal character name	U+nnnn	Exactly 4 hexadecimal digits. Use the \Unnnnnnnn notation for astral characters.
`\Unnnnnnnn`	universal character name	U+nnnnnnnn	Exactly 8 hexadecimal digits.

A backslash (\) followed by another character is simply preserved as is.

Octal and hexadecimal escape sequences can represent ASCII characters (ordinal value 0 to 127) or the UTF-8 encoding of non-ASCII characters.

A string literal can contain printable ASCII characters (including the escape sequences presented above) and non-ASCII characters; non-printable ASCII characters (such as an unescaped tab) are not allowed.

Here are some examples:

Slice

```
const string AnOrdinaryString = "Hello World!";

const string DoubleQuote =      "\"";
const string TwoSingleQuotes =  "'\'";     // ' and \' are OK
const string QuestionMark =     "\?";
const string Backslash =        "\\";
const string AudibleBell =      "\a";
const string Backspace =        "\b";
const string FormFeed =         "\f";
const string Newline =          "\n";
const string CarriageReturn =   "\r";
const string HorizontalTab =    "\t";
const string VerticalTab =      "\v";

const string OctalEscape =    "\007";     // Same as \a
const string HexEscape1  =    "\x07";     // Ditto
const string HexEscape2 =     "\x41F";    // Same as AF
const string Universal1 =     "\u0041";   // Same as A
const string Universal2 = "\U00000041";   // Ditto

const string EuroSign1 =          "€";   // Euro sign (U+20AC)
const string EuroSign2 =      "\u20AC";   // Euro sign as a short
universal character name
const string EuroSign3 =  "\U000020ac";   // Euro sign as a long
universal character name
const string EuroSign4 = "\xe2\x82\xAC";  // Euro sign in UTF-8
encoding, using hex escape sequences
const string EuroSign5 = "\342\202\254";  // Euro sign in UTF-8
encoding, using octal escape sequences
const string EuroSign6 = "\342\x82\254";  // Euro sign in UTF-8
encoding, using a mix or hex and octal escape sequences
```

Slice

```
const string NullString = null;    // Illegal!
```

Null strings simply do not exist in Slice and, therefore, do not exist as a legal value for a string anywhere in the Ice platform. The reason for this decision is that null strings do not exist in many programming languages.

Constant Expressions

A constant definition may also refer to another constant. It is not necessary for both constants to have the same Slice type, but the value of the existing constant must be compatible with the type of the constant being defined.

Consider the examples below:

Slice

```
const int SIZE = 500;

const int DEFAULT_SIZE = SIZE; // OK
const short SHORT_SIZE = SIZE; // OK
const byte BYTE_SIZE = SIZE;   // ERROR
```

The `DEFAULT_SIZE` constant is legal because it has the same type as `SIZE`, and `SHORT_SIZE` is legal because the value of `SIZE` (`500`) is within the range of the Slice `short` type. However, `BYTE_SIZE` is illegal because the value of `SIZE` is outside the range of the `byte` type.

See Also

- Enumerations
- Structures
- Sequences
- Dictionaries

Interfaces, Operations, and Exceptions

The central focus of Slice is on defining interfaces, for example:

Slice

```
module M
{
    struct TimeOfDay
    {
        short hour;         // 0 - 23
        short minute;       // 0 - 59
        short second;       // 0 - 59
    }

    interface Clock
    {
        TimeOfDay getTime();
        void setTime(TimeOfDay time);
    }
}
```

This definition defines an interface type called `Clock`. The interface supports two operations: `getTime` and `setTime`. Clients access an object supporting the `Clock` interface by invoking an operation on the proxy for the object: to read the current time, the client invokes the `getTime` operation; to set the current time, the client invokes the `setTime` operation, passing an argument of type `TimeOfDay`.

Invoking an operation on a proxy instructs the Ice run time to send a message to the target object. The target object can be in another address space or can be collocated (in the same process) as the caller — the location of the target object is transparent to the client. If the target object is in another (possibly remote) address space, the Ice run time invokes the operation via a remote procedure call; if the target is collocated with the client, the Ice run time bypasses the network stack altogether to deliver the request more efficiently.

You can think of an interface definition as the equivalent of the public part of a C++ class definition or as the equivalent of a Java interface, and of operation definitions as (virtual) member functions. Note that nothing but operation definitions are allowed to appear inside an interface definition. In particular, you cannot define a type, an exception, or a data member inside an interface. This does not mean that your object implementation cannot contain state — it can, but how that state is implemented (in the form of data members or otherwise) is hidden from the client and, therefore, need not appear in the object's interface definition.

An Ice object has exactly one (most derived) Slice interface type. Of course, you can create multiple Ice objects that have the same type; to draw the analogy with C++, a Slice interface corresponds to a C++ class definition, whereas an Ice object corresponds to a C++ class instance (but Ice objects can be implemented in multiple different address spaces).

Ice also provides multiple interfaces for the same Ice object via a feature called *facets*.

A Slice interface defines the smallest grain of distribution in Ice: each Ice object has a unique identity (encapsulated in its proxy) that distinguishes it from all other Ice objects; for communication to take place, you must invoke operations on an object's proxy. There is no other notion of an addressable entity in Ice. You cannot, for example, instantiate a Slice structure and have clients manipulate that structure remotely. To make the structure accessible, you must create an interface that allows clients to access the structure.

The partition of an application into interfaces therefore has profound influence on the overall architecture. Distribution boundaries must follow interface boundaries; you can spread the implementation of interfaces over multiple address spaces (and you can implement multiple interfaces in the same address space), but you cannot implement parts of interfaces in different address spaces.

Topics

See Also

- Classes
- Versioning

Operations

On this page:

Parameters and Return Values

An operation definition must contain a return type and zero or more parameter definitions. For example, in the `Clock` interface, the `getTime` operation has a return type of `TimeOfDay` and the `setTime` operation has a return type of `void`. You must use `void` to indicate that an operation returns no value — there is no default return type for Slice operations.

An operation can have one or more input parameters. For example, `setTime` accepts a single input parameter of type `TimeOfDay` called `time`. Of course, you can use multiple input parameters:

Slice

```
module M
{
    interface CircadianRhythm
    {
        void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
        // ...
    }
}
```

Note that the parameter name (as for Java) is mandatory. You cannot omit the parameter name, so the following is in error:

Slice

```
module M
{
    interface CircadianRhythm
    {
        void setSleepPeriod(TimeOfDay, TimeOfDay);  // Error!
        // ...
    }
}
```

By default, parameters are sent from the client to the server, that is, they are input parameters. To pass a value from the server to the client, you can use an output parameter, indicated by the `out` keyword. For example, an alternative way to define the `getTime` operation in the `Clock` interface would be:

Slice

```
void getTime(out TimeOfDay time);
```

This achieves the same thing but uses an output parameter instead of the return value. As with input parameters, you can use multiple

output parameters:

Slice

```
module M
{
    interface CircadianRhythm
    {
        void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
        void getSleepPeriod(out TimeOfDay startTime,
out TimeOfDay stopTime);
        // ...
    }
}
```

If you have both input and output parameters for an operation, the output parameters must follow the input parameters:

Slice

```
void changeSleepPeriod(    TimeOfDay startTime,
TimeOfDay stopTime,     // OK
                       out TimeOfDay prevStartTime,
out TimeOfDay prevStopTime);

void changeSleepPeriod(out TimeOfDay prevStartTime, out TimeOfDay prevS
topTime, // Error
                           TimeOfDay startTime,
TimeOfDay stopTime);
```

Slice does not support parameters that are both input and output parameters (call by reference). The reason is that, for remote calls, reference parameters do not result in the same savings that one can obtain for call by reference in programming languages. (Data still needs to be copied in both directions and any gains in marshaling efficiency are negligible.) Also, reference (or input-output) parameters result in more complex language mappings, with concomitant increases in code size.

Optional Parameters and Return Values

An operation's return value and parameters may be declared as optional to indicate that a program can leave their values unset. Parameters not declared as optional are known as *required* parameters; a program must supply legal values for all required parameters. In the discussion below, we use *parameter* to refer to input parameters, output parameters, and return values.

A unique, non-negative integer *tag* must be assigned to each optional parameter:

Slice

```
optional(3) bool example(optional(2) string name, out optional(1) int
value);
```

The scope of a tag is limited to its operation and has no effect on other operations.

An operation's signature can include any combination of required and optional parameters, but output parameters still must follow input

parameters:

Slice

```
bool example(string name, optional(3) string referrer, out optional(1)
string promo, out int id);
```

Language mappings specify an API for passing optional parameters and testing whether a parameter is present. Refer to the language mapping sections for more details on the optional parameter API.

> A well-behaved program must test for the presence of an optional parameter and not assume that it is always set. Dereferencing an unset optional parameter causes a run-time error.

Style of Operation Definition

As you would expect, language mappings follow the style of operation definition you use in Slice: Slice return types map to programming language return types, and Slice parameters map to programming language parameters.

For operations that return only a single value, it is common to return the value from the operation instead of using an out-parameter. This style maps naturally into all programming languages. Note that, if you use an out-parameter instead, you impose a different API style on the client: most programming languages permit the return value of a function to be ignored whereas it is typically not possible to ignore an output parameter.

For operations that return multiple values, it is common to return all values as out-parameters and to use a return type of `void`. However, the rule is not all that clear-cut because operations with multiple output values can have one particular value that is considered more "important" than the remainder. A common example of this is an iterator operation that returns items from a collection one-by-one:

Slice

```
bool next(out RecordType r);
```

The `next` operation returns two values: the record that was retrieved and a Boolean to indicate the end-of-collection condition. (If the return value is `false`, the end of the collection has been reached and the parameter `r` has an undefined value.) This style of definition can be useful because it naturally fits into the way programmers write control structures. For example:

```
while(next(record))
{
    // Process record...
}

if(next(record))
{
    // Got a valid record...
}
```

Overloading Operations

Slice does not support any form of overloading of operations. For example:

Slice

```
interface CircadianRhythm
{
    void modify(TimeOfDay startTime, TimeOfDay endTime);
    void modify(    TimeOfDay startTime,        // Error
                    TimeOfDay endTime,
                out timeOfDay prevStartTime,
                out TimeOfDay prevEndTime);
}
```

Operations in the same interface must have different names, regardless of what type and number of parameters they have. This restriction exists because overloaded functions cannot sensibly be mapped to languages without built-in support for overloading.

> Name mangling is not an option in this case: while it works fine for compilers, it is unacceptable to humans.

Idempotent Operations

Some operations, such as `getTime` in the `Clock` interface, do not modify the state of the object they operate on. They are the conceptual equivalent of C++ `const` member functions. Similary, `setTime` does modify the state of the object, but is idempotent. You can indicate this in Slice as follows:

Slice

```
interface Clock
{
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
}
```

This marks the `getTime` and `setTime` operations as idempotent. An operation is idempotent if two successive invocations of the operation have the same effect as a single invocation. For example, `x = 1;` is an idempotent operation because it does not matter whether it is executed once or twice — either way, `x` ends up with the value 1. On the other hand, `x += 1;` is not an idempotent operation because executing it twice results in a different value for `x` than executing it once. Obviously, any read-only operation is idempotent.

The `idempotent` keyword is useful because it allows the Ice run time to be more aggressive when performing automatic retries to recover from errors. Specifically, Ice guarantees *at-most-once* semantics for operation invocations:

- For normal (not idempotent) operations, the Ice run time has to be conservative about how it deals with errors. For example, if a client sends an operation invocation to a server and then loses connectivity, there is no way for the client-side run time to find out whether the request it sent actually made it to the server. This means that the run time cannot attempt to recover from the error by re-establishing a connection and sending the request a second time because that could cause the operation to be invoked a second time and violate at-most-once semantics; the run time has no option but to report the error to the application.
- For `idempotent` operations, on the other hand, the client-side run time can attempt to re-establish a connection to the server and safely send the failed request a second time. If the server can be reached on the second attempt, everything is fine and the application never notices the (temporary) failure. Only if the second attempt fails need the run time report the error back to the application. (The number of retries can be increased with an Ice configuration parameter.)

See Also

- Interfaces, Operations, and Exceptions
- User Exceptions
- Run-Time Exceptions

User Exceptions

On this page:

User Exception Syntax and Semantics

Looking at the `setTime` operation in the `Clock` interface, we find a potential problem: given that the `TimeOfDay` structure uses `short` as the type of each field, what will happen if a client invokes the `setTime` operation and passes a `TimeOfDay` value with meaningless field values, such as `-199` for the minute field, or `42` for the hour? Obviously, it would be nice to provide some indication to the caller that this is meaningless. Slice allows you to define user exceptions to indicate error conditions to the client. For example:

Slice

```
module M
{
    exception Error {} // Empty exceptions are legal

    exception RangeError
    {
        TimeOfDay errorTime;
        TimeOfDay minTime;
        TimeOfDay maxTime;
    }
}
```

A user exception is much like a structure in that it contains a number of data members. However, unlike structures, exceptions can have zero data members, that is, be empty. Like classes, user exceptions support inheritance and may include optional data members.

Default Values for User Exception Members

You can specify a default value for an exception data member that has one of the following types:

- An integral type (`byte`, `short`, `int`, `long`)
- A floating point type (`float` or `double`)
- `string`
- `bool`
- `enum`

For example:

Slice

```
module M
{
    exception RangeError
    {
        TimeOfDay errorTime;
        TimeOfDay minTime;
        TimeOfDay maxTime;
        string reason = "out of range";
    }
}
```

The legal syntax for literal values is the same as for Slice constants, and you may also use a constant as a default value. The language mapping guarantees that data members are initialized to their declared default values using a language-specific mechanism.

Declaring User Exceptions in Operations

Exceptions allow you to return an arbitrary amount of error information to the client if an error condition arises in the implementation of an operation. Operations use an exception specification to indicate the exceptions that may be returned to the client:

Slice

```
module M
{
    interface Clock
    {
        idempotent TimeOfDay getTime();
        idempotent void setTime(TimeOfDay time)
            throws RangeError, Error;
    }
}
```

This definition indicates that the `setTime` operation may throw either a `RangeError` or an `Error` user exception (and no other type of exception). If the client receives a `RangeError` exception, the exception contains the `TimeOfDay` value that was passed to `setTime` and caused the error (in the `errorTime` member), as well as the minimum and maximum time values that can be used (in the `minTime` and `maxTime` members). If `setTime` failed because of an error not caused by an illegal parameter value, it throws `Error`. Obviously, because `Error` does not have data members, the client will have no idea what exactly it was that went wrong — it simply knows that the operation did not work.

To indicate that an operation does not throw any user exception, simply omit the exception specification. (There is no empty exception specification in Slice.)

As of Ice 3.7, the server-side Ice run time does not verify that a user exception raised by an operation is compatible with the exceptions listed in its Slice definition, although your implementation language may enforce its own restrictions. The Ice run time in the client does validate user exceptions and raises `UnknownUserException` if it receives an unexpected user exception.

Restrictions for User Exceptions

Exceptions are not first-class data types and first-class data types are not exceptions:

- You cannot pass an exception as a parameter value.
- You cannot use an exception as the type of a data member.

- You cannot use an exception as the element type of a sequence.
- You cannot use an exception as the key or value type of a dictionary.
- You cannot throw a value of non-exception type (such as a value of type `int` or `string`).

The reason for these restrictions is that some implementation languages use a specific and separate type for exceptions (in the same way as Slice does). For such languages, it would be difficult to map exceptions if they could be used as an ordinary data type. (C++ is somewhat unusual among programming languages by allowing arbitrary types to be used as exceptions.)

User Exception Inheritance

Exceptions support inheritance. For example:

Slice

```
exception ErrorBase
{
    string reason;
}

enum RTError
{
    DivideByZero, NegativeRoot, IllegalNull /* ... */
}

exception RuntimeError extends ErrorBase
{
    RTError err;
}

enum LError { ValueOutOfRange, ValuesInconsistent, /* ... */ }

exception LogicError extends ErrorBase
{
    LError err;
}

exception RangeError extends LogicError
{
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
}
```

These definitions set up a simple exception hierarchy:

- `ErrorBase` is at the root of the tree and contains a string explaining the cause of the error.
- Derived from `ErrorBase` are `RuntimeError` and `LogicError`. Each of these exceptions contains an enumerated value that further categorizes the error.
- Finally, `RangeError` is derived from `LogicError` and reports the details of the specific error.

Setting up exception hierarchies such as this not only helps to create a more readable specification because errors are categorized, but also can be used at the language level to good advantage. For example, the Slice C++ mapping preserves the exception hierarchy so you can catch exceptions generically as a base exception, or set up exception handlers to deal with specific exceptions.

Looking at the exception hierarchy, it is not clear whether, at run time, the application will only throw most derived exceptions, such as `Rang`

`eError`, or if it will also throw base exceptions, such as `LogicError`, `RuntimeError`, and `ErrorBase`. If you want to indicate that a base exception, interface, or class is abstract (will not be instantiated), you can add a comment to that effect.

Note that, if the exception specification of an operation indicates a specific exception type, at run time, the implementation of the operation may also throw more derived exceptions. For example:

Slice

```
exception Base
{
    // ...
}

exception Derived extends Base
{
    // ...
}

interface Example
{
    void op() throws Base;      // May throw Base or Derived
}
```

In this example, `op` may throw a `Base` or a `Derived` exception, that is, any exception that is compatible with the exception types listed in the exception specification can be thrown at run time.

As a system evolves, it is quite common for new, derived exceptions to be added to an existing hierarchy. Assume that we initially construct clients and server with the following definitions:

Slice

```
exception Error
{
    // ...
}

interface Application
{
    void doSomething() throws Error;
}
```

Also assume that a large number of clients are deployed in field, that is, when you upgrade the system, you cannot easily upgrade all the clients. As the application evolves, a new exception is added to the system and the server is redeployed with the new definition:

Slice

```
exception Error
{
    // ...
}

exception FatalApplicationError extends Error
{
    // ...
}

interface Application
{
    void doSomething() throws Error;
}
```

This raises the question of what should happen if the server throws a `FatalApplicationError` from `doSomething`. The answer depends whether the client was built using the old or the updated definition:

- If the client was built using the same definition as the server, it simply receives a `FatalApplicationError`.
- If the client was built with the original definition, that client has no knowledge that `FatalApplicationError` even exists. In this case, the Ice run time automatically slices the exception to the most-derived type that is understood by the receiver (`Error`, in this case) and discards the information that is specific to the derived part of the exception. (This is exactly analogous to catching C++ exceptions by value — the exception is sliced to the type used in the `catch`-clause.)

Exceptions support single inheritance only. (Multiple inheritance would be difficult to map into many programming languages.)

See Also

- Constants and Literals
- Operations
- Run-Time Exceptions
- Proxies for Ice Objects
- Interface Inheritance
- Optional Data Members

Run-Time Exceptions

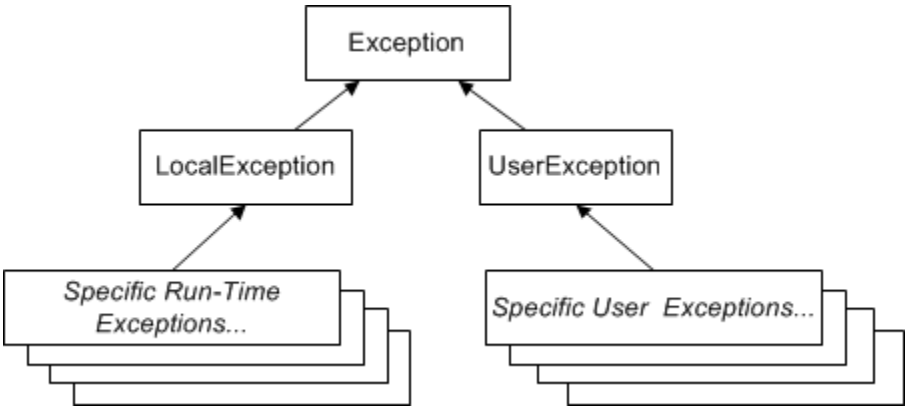
In addition to any user exceptions that are listed in an operation's exception specification, an operation can also throw Ice *run-time exceptions*. Run-time exceptions are predefined exceptions that indicate platform-related run-time errors. For example, if a networking error interrupts communication between client and server, the client is informed of this by a run-time exception, such as `ConnectTimeoutException` or `SocketException`.

The exception specification of an operation must not list any run-time exceptions. (It is understood that all operations can raise run-time exceptions and you are not allowed to restate that.)

On this page:

Inheritance Hierarchy for Exceptions

All the Ice run-time and user exceptions are arranged in an inheritance hierarchy, as shown below:

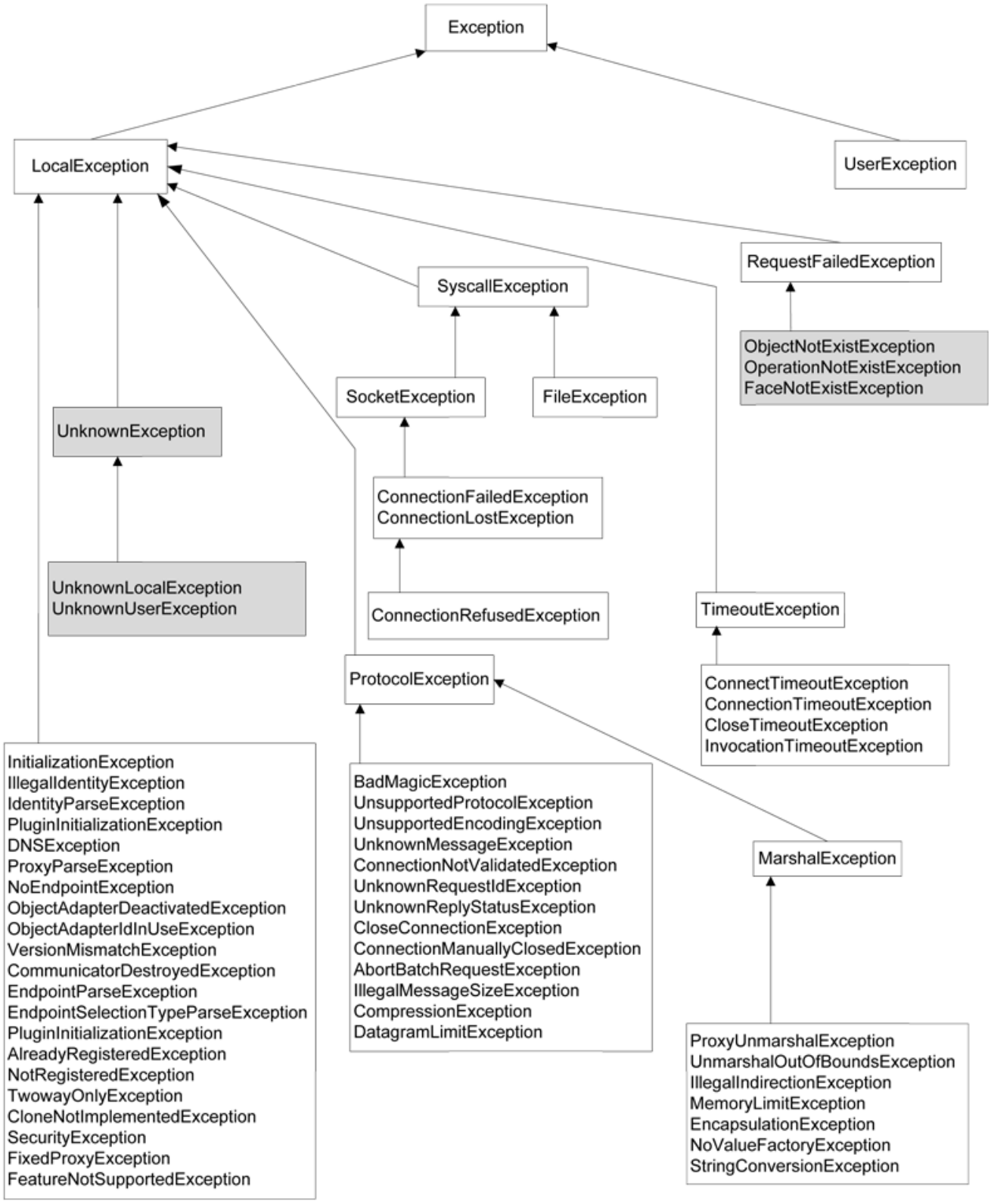


Inheritance structure for exceptions.

`Ice::Exception` is at the root of the inheritance hierarchy. Derived from that are the (abstract) types `Ice::LocalException` and `Ice::UserException`. In turn, all run-time exceptions are derived from `Ice::LocalException`, and all user exceptions are derived from `Ice::UserException`.

> Ice run-time exceptions are all defined in Slice as `local exceptions`. *Local exception* is a synonym for Ice run-time exception.

This figures shows the complete hierarchy of the Ice run-time exceptions:



Ice run-time exception hierarchy. (Shaded exceptions can be sent by the server.)

Note that Ice run-time exception hierarchy groups several exceptions into a single box to save space (which, strictly, is incorrect UML syntax). Also note that some run-time exceptions have data members, which, for brevity, we have omitted in the Ice run-time exception hierarchy. These data members provide additional information about the precise cause of an error.

Many of the run-time exceptions have self-explanatory names, such as `MemoryLimitException`. Others indicate problems in the Ice run time, such as `EncapsulationException`. Still others can arise only through application programming errors, such as `TwowayOnlyException`. In practice, you will likely never see most of these exceptions. However, there are a few run-time exceptions you will encounter and whose meaning you should know.

Local Versus Remote Exceptions

Most error conditions are detected on the client side and raised locally in the client. For example, if an attempt to contact a server fails, the client-side run time raises a `ConnectTimeoutException`.

However, there are a few specific error conditions (shown as shaded in the Ice run-time exception hierarchy diagram) that are detected by the server and transmitted to the client via the Ice protocol: `ObjectNotExistException`, `FacetNotExistException` and `OperationNotExistException` (collectively the Request Failed exceptions) plus `UnknownException`, `UnknownLocalException` and `UnknownUserException` (collectively the Unknown exceptions).

All other run-time exceptions (not shaded in the Ice run-time exception hierarchy) are detected by the client-side run time and are raised locally.

It is possible for the implementation of an operation to throw Ice run-time exceptions (as well as user exceptions). For example, if a client holds a proxy to an object that no longer exists in the server, your server application code is required to throw an `ObjectNotExistException`. If you do throw run-time exceptions from your application code, you should take care to throw a run-time exception only if appropriate, that is, do not use run-time exceptions to indicate something that really should be a user exception. Doing so can be very confusing to the client: if the application "hijacks" some run-time exceptions for its own purposes, the client can no longer decide whether the exception was thrown by the Ice run time or by the server application code. This can make debugging very difficult.

Request Failed Exceptions

`ObjectNotExistException`

This exception indicates that a request was delivered to the server but the server could not locate a servant with the identity that is embedded in the proxy. In other words, the server could not find an object to dispatch the request to.

> The Ice run time raises `ObjectNotExistException` only if there are no facets in existence with a matching identity; otherwise, it raises `FacetNotExistException`.

Most likely, this is the case because the object existed some time in the past and has since been destroyed, but the same exception is also raised if a client uses a proxy with the identity of an object that has never been created.

`FacetNotExistException`

The client attempted to contact a non-existent facet of an object, that is, the server has at least one servant with the given identity, but no servant with a matching facet name.

`OperationNotExistException`

This exception is raised if the server could locate an object with the correct identity but, on attempting to dispatch the client's operation invocation, the server found that the target object does not have such an operation. You will see this exception in only two cases:

- Client and server have been built with Slice definitions for an interface that disagree with each other, that is, the client was built with an interface definition for the object that indicates that an operation exists, but the server was built with a different version of the interface definition in which the operation is absent.
- You have used an unchecked down-cast on a proxy of the incorrect type.

> `ObjectNotExistException`, `FacetNotExistException` and `OperationNotExistException` derive from `RequestFailedException`, and don't add any data member for this exception:

Slice

```
module Ice
{
    local slice RequestFailException
    {
        Identity id;
        string facet;
        string operation;
    }
}
```

`RequestFailedException` itself is not transmissible as a response from a server to client—only the three derived exceptions are.

If you throw one of these three exceptions from the implementation of an operation, and you leave `id`, `facet` or `operation` empty, Ice will automatically fill-in the missing data members using values from Current.

Unknown Exceptions

Any error condition on the server side that is not described by one of the three preceding exceptions is made known to the client as one of three generic exceptions: `UnknownUserException`, `UnknownLocalException`, or `UnknownException`. Furthermore if a servant implementation throws one of these `Unknown` exceptions, the Ice run time transmits it as is–it does not wrap it a new `UnknownLocalException`.

```
module Ice
{
    local exception UnknownException
    {
        string unknown;
    }

    local exception UnknownLocalException extends UnknownException
    {
    }

    local exception UnknownUserException extends UnknownException
    {
    }
}
```

UnknownUserException

This exception indicates that an operation implementation has thrown a Slice exception that is not declared in the operation's exception specification (and is not derived from one of the exceptions in the operation's exception specification). Ice itself never throws this exception, as all user-exception checking for a given operation is performed only in the client's generated code and Ice run-time. It is nevertheless permissible for a servant to throw this exception.

`UnknownLocalException`

If an operation implementation raises a run-time exception other than `ObjectNotExistException`, `FacetNotExistException`, `Opera`

`tionNotExistException` or `UnknownException` (such as a `NotRegisteredException`), the client receives an `UnknownLocalException`. In other words, the Ice protocol does not transmit the exact exception that was encountered in the server, but simply returns a bit to the client in the reply to indicate that the server encountered a run-time exception.

A common cause for a client receiving an `UnknownLocalException` is failure to catch and handle all exceptions in the server. For example, if the implementation of an operation encounters an exception it does not handle, the exception propagates all the way up the call stack until the stack is unwound to the point where the Ice run time invoked the operation. The Ice run time catches all Ice exceptions that "escape" from an operation invocation and returns them to the client as an `UnknownLocalException`.

`UnknownException`

An operation has thrown a non-Ice exception. For example, if the operation in the server throws a C++ exception, such as a `std::bad_alloc`, or a Java exception, such as a `ClassCastException`, the client receives an `UnknownException`.

See Also

- User Exceptions
- Interfaces, Operations, and Exceptions
- Operations
- Proxies for Ice Objects
- Interface Inheritance
- Versioning

Proxies for Ice Objects

Building on the `Clock` example, we can create definitions for a world-time server:

Slice

```
module M
{
    exception GenericError
    {
        string reason;
    }

    struct TimeOfDay
    {
        short hour;         // 0 - 23
        short minute;       // 0 - 59
        short second;       // 0 - 59
    }

    exception BadTimeVal extends GenericError {}

    interface Clock
    {
        idempotent TimeOfDay getTime();
        idempotent void setTime(TimeOfDay time) throws BadTimeVal;
    }

    dictionary<string, Clock*> TimeMap; // Time zone name to clock map

    exception BadZoneName extends GenericError {}

    interface WorldTime
    {
        idempotent void addZone(string zoneName, Clock* zoneClock);
        void removeZone(string zoneName) throws BadZoneName;
        idempotent Clock* findZone(string zoneName) throws BadZoneName;
        idempotent TimeMap listZones();
        idempotent void setZones(TimeMap zones);
    }
}
```

The `WorldTime` interface acts as a collection manager for clocks, one for each time zone. In other words, the `WorldTime` interface manages a collection of pairs. The first member of each pair is a time zone name; the second member of the pair is the clock that provides the time for that zone. The interface contains operations that permit you to add or remove a clock from the map (`addZone` and `removeZone`), to search for a particular time zone by name (`findZone`), and to read or write the entire map (`listZones` and `setZones`).

The `WorldTime` example illustrates an important Slice concept: note that `addZone` accepts a parameter of type `Clock*` and `findZone` returns a parameter of type `Clock*`. In other words, interfaces are types in their own right and can be passed as parameters. The `*` operator is known as the *proxy operator*. Its left-hand argument must be an interface (or class) and its return type is a proxy. A proxy is like a pointer that can denote an object. The semantics of proxies are very much like those of C++ class instance pointers:

- A proxy can be null.

- A proxy can dangle (point at an object that is no longer there).
- Operations dispatched via a proxy use late binding: if the actual run-time type of the object denoted by the proxy is more derived than the proxy's type, the implementation of the most-derived interface will be invoked.

When a client passes a `Clock` proxy to the `addZone` operation, the proxy denotes an actual `Clock` object in a server. The `Clock` *Ice object* denoted by that proxy may be implemented in the same server process as the `WorldTime` interface, or in a different server process. Where the `Clock` object is physically implemented matters neither to the client nor to the server implementing the `WorldTime` interface; if either invokes an operation on a particular clock, such as `getTime`, an RPC call is sent to whatever server implements that particular clock. In other words, a proxy acts as a local "ambassador" for the remote object; invoking an operation on the proxy forwards the invocation to the actual object implementation. If the object implementation is in a different address space, this results in a remote procedure call; if the object implementation is collocated in the same address space, the Ice run time may optimize the invocation.

Note that proxies also act very much like pointers in their sharing semantics: if two clients have a proxy to the same object, a state change made by one client (such as setting the time) will be visible to the other client.

Proxies are strongly typed (at least for statically typed languages, such as C++ and Java). This means that you cannot pass something other than a `Clock` proxy to the `addZone` operation; attempts to do so are rejected at compile time.

See Also

- Classes
- Interfaces, Operations, and Exceptions
- User Exceptions
- Run-Time Exceptions
- Interface Inheritance

Interface Inheritance

On this page:

Interface Inheritance

Interfaces support inheritance. For example, we could extend our world-time server to support the concept of an alarm clock:

Slice

```
module M
{
    interface AlarmClock extends Clock
    {
        idempotent TimeOfDay getAlarmTime();
        idempotent void setAlarmTime(TimeOfDay alarmTime)
            throws BadTimeVal;
    }
}
```

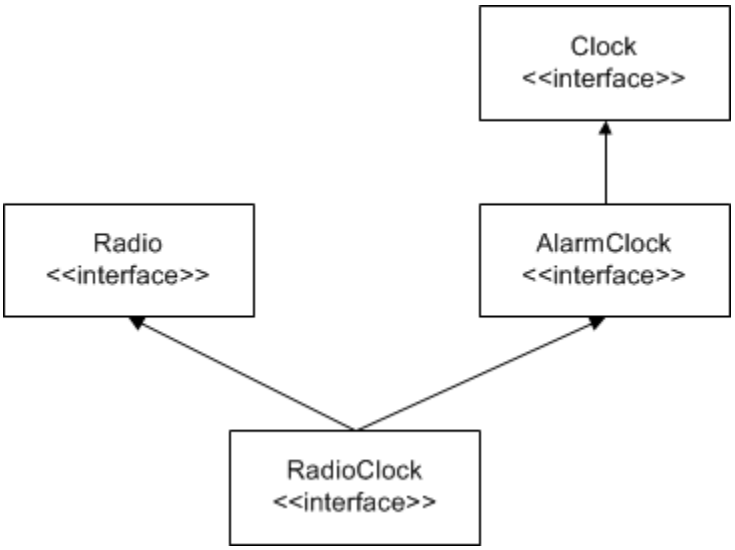
The semantics of this are the same as for C++ or Java: `AlarmClock` is a subtype of `Clock` and an `AlarmClock` proxy can be substituted wherever a `Clock` proxy is expected. Obviously, an `AlarmClock` supports the same `getTime` and `setTime` operations as a `Clock` but also supports the `getAlarmTime` and `setAlarmTime` operations.

Multiple interface inheritance is also possible. For example, we can construct a radio alarm clock as follows:

Slice

```
module M
{
    interface Radio
    {
        void setFrequency(long hertz) throws GenericError;
        void setVolume(long dB) throws GenericError;
    }

    enum AlarmMode { RadioAlarm, BeepAlarm }

    interface RadioClock extends Radio, AlarmClock
    {
        void setMode(AlarmMode mode);
        AlarmMode getMode();
    }
}
```

`RadioClock` extends both `Radio` and `AlarmClock` and can therefore be passed where a `Radio`, an `AlarmClock`, or a `Clock` is expected. The inheritance diagram for this definition looks as follows:



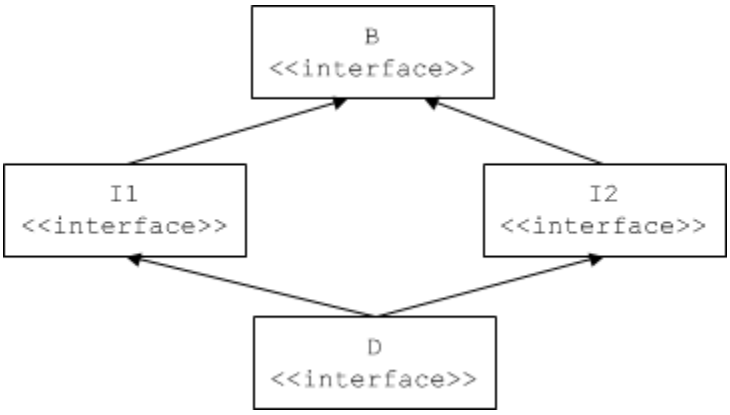
Inheritance diagram for `RadioClock`.

Interfaces that inherit from more than one base interface may share a common base interface. For example, the following definition is legal:

Slice

```
interface B { /* ... */ }
interface I1 extends B { /* ... */ }
interface I2 extends B { /* ... */ }
interface D extends I1, I2 { /* ... */ }
```

This definition results in the familiar diamond shape:



Diamond-shaped inheritance.

Interface Inheritance Limitations

If an interface uses multiple inheritance, it must not inherit the same operation name from more than one base interface. For example, the following definition is illegal:

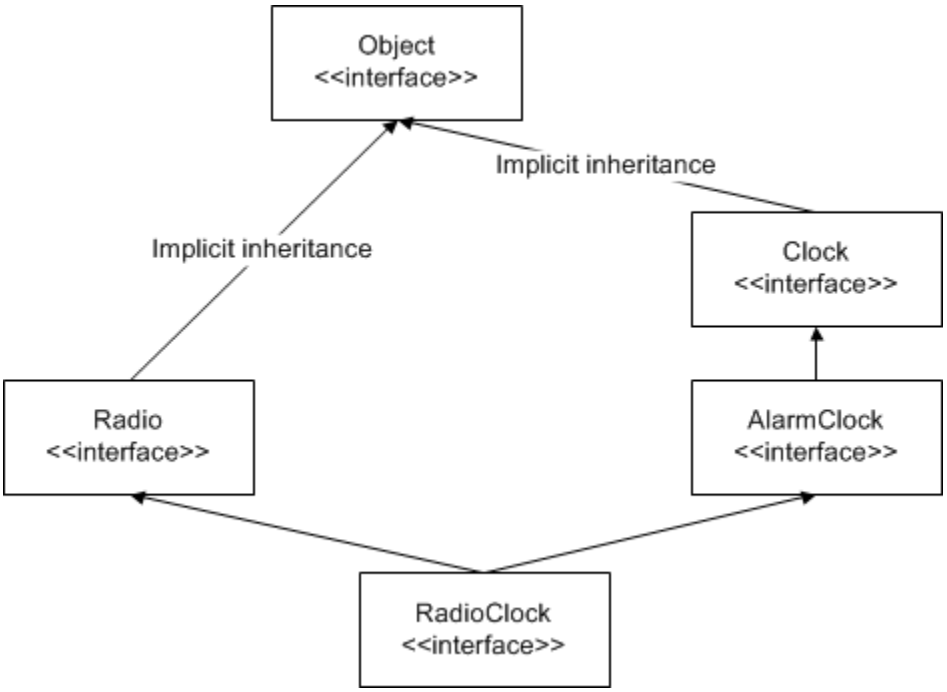
Slice

```
interface Clock
{
    void set(TimeOfDay time);                   // set time
}

interface Radio
{
    void set(long hertz);                       // set frequency
}

interface RadioClock extends Radio, Clock     // Illegal!
{
    // ...
}
```

This definition is illegal because `RadioClock` inherits two `set` operations, `Radio::set` and `Clock::set`. The Slice compiler makes this illegal because (unlike C++) many programming languages do not have a built-in facility for disambiguating the different operations. In Slice, the simple rule is that all inherited operations must have unique names. (In practice, this is rarely a problem because inheritance is rarely added to an interface hierarchy "after the fact". To avoid accidental clashes, we suggest that you use descriptive operation names, such as `setTime` and `setFrequency`. This makes accidental name clashes less likely.)

Implicit Inheritance from Object

All Slice interfaces are ultimately derived from `Object`. For example, the inheritance hierarchy would be shown more correctly as:



Implicit inheritance from `Object`.

Because all interfaces have a common base interface, we can pass any type of interface as that type. For example:

Slice

```
interface ProxyStore
{
    idempotent void putProxy(string name, Object* o);
    idempotent Object* getProxy(string name);
}
```

`Object` is a Slice keyword (note the capitalization) that denotes the root type of the inheritance hierarchy. The `ProxyStore` interface is a generic proxy storage facility: the client can call `putProxy` to add a proxy of any type under a given name and later retrieve that proxy again by calling `getProxy` and supplying that name. The ability to generically store proxies in this fashion allows us to build general-purpose facilities, such as a naming service that can store proxies and deliver them to clients. Such a service, in turn, allows us to avoid hard-coding proxy details into clients and servers.

Inheritance from type `Object` is always implicit. For example, the following Slice definition is illegal:

Slice

```
interface MyInterface extends Object { /* ... */ } // Error!
```

It is understood that all interfaces inherit from type `Object`; you are not allowed to restate that.

Type `Object` is mapped to an abstract type by the various language mappings, so you cannot instantiate an Ice object of that type.

Null Proxies

Looking at the `ProxyStore` interface once more, we notice that `getProxy` does not have an exception specification. The question then is what should happen if a client calls `getProxy` with a name under which no proxy is stored? Obviously, we could add an exception to indicate this condition to `getProxy`. However, another option is to return a *null proxy*. Ice has the built-in notion of a null proxy, which is a proxy that "points nowhere". When such a proxy is returned to the client, the client can test the value of the returned proxy to check whether it is null or denotes a valid object.

A more interesting question is: "which approach is more appropriate, throwing an exception or returning a null proxy?" The answer depends on the expected usage pattern of an interface. For example, if, in normal operation, you do not expect clients to call `getProxy` with a non-existent name, it is better to throw an exception. (This is probably the case for our `ProxyStore` interface: the fact that there is no `list` operation makes it clear that clients are expected to know which names are in use.)

On the other hand, if you expect that clients will occasionally try to look up something that is not there, it is better to return a null proxy. The reason is that throwing an exception breaks the normal flow of control in the client and requires special handling code. This means that you should throw exceptions only in exceptional circumstances. For example, throwing an exception if a database lookup returns an empty result set is wrong; it is expected and normal that a result set is occasionally empty.

It is worth paying attention to such design issues: well-designed interfaces that get these details right are easier to use and easier to understand. Not only do such interfaces make life easier for client developers, they also make it less likely that latent bugs cause problems later.

Self-Referential Interfaces

Proxies have pointer semantics, so we can define self-referential interfaces. For example:

Slice

```
interface Link
{
    idempotent SomeType getValue();
    idempotent Link* next();
}
```

The `Link` interface contains a `next` operation that returns a proxy to a `Link` interface. Obviously, this can be used to create a chain of interfaces; the final link in the chain returns a null proxy from its `next` operation.

Empty Interfaces

The following Slice definition is legal:

Slice

```
interface Empty {}
```

The Slice compiler will compile this definition without complaint. An interesting question is: "why would I need an empty interface?" In most cases, empty interfaces are an indication of design errors. Here is one example:

Slice

```
interface ThingBase {}

interface Thing1 extends ThingBase
{
    // Operations here...
}

interface Thing2 extends ThingBase
{
    // Operations here...
}
```

Looking at this definition, we can make two observations:

- `Thing1` and `Thing2` have a common base and are therefore related.
- Whatever is common to `Thing1` and `Thing2` can be found in interface `ThingBase`.

Of course, looking at `ThingBase`, we find that `Thing1` and `Thing2` do not share any operations at all because `ThingBase` is empty. Given that we are using an object-oriented paradigm, this is definitely strange: in the object-oriented model, the *only* way to communicate with an object is to send a message to the object. But, to send a message, we need an operation. Given that `ThingBase` has no operations, we cannot send a message to it, and it follows that `Thing1` and `Thing2` are *not* related because they have no common operations. But of course, seeing that `Thing1` and `Thing2` have a common base, we conclude that they *are* related, otherwise the common base would not exist. At this point, most programmers begin to scratch their head and wonder what is going on here.

One common use of the above design is a desire to treat `Thing1` and `Thing2` polymorphically. For example, we might continue the previous definition as follows:

Slice

```
interface ThingUser
{
    void putThing(ThingBase* thing);
}
```

Now the purpose of having the common base becomes clear: we want to be able to pass both `Thing1` and `Thing2` proxies to `putThing`. Does this justify the empty base interface? To answer this question, we need to think about what happens in the implementation of `putThing`. Obviously, `putThing` cannot possibly invoke an operation on a `ThingBase` because there are no operations. This means that `putThing` can do one of two things:

1. putThing can simply remember the value of `thing`.
2. putThing can try to down-cast to either `Thing1` or `Thing2` and then invoke an operation. The pseudo-code for the implementation of `putThing` would look something like this:

```
void putThing(ThingBase thing)
{
    if(is_a(Thing1, thing))
    {
        // Do something with Thing1...
    }
    else if(is_a(Thing2, thing))
    {
        // Do something with Thing2...
    }
    else
    {
        // Might be a ThingBase?
        // ...
    }
}
```

   The implementation tries to down-cast its argument to each possible type in turn until it has found the actual run-time type of the argument. Of course, any object-oriented text book worth its price will tell you that this is an abuse of inheritance and leads to maintenance problems.

If you find yourself writing operations such as `putThing` that rely on artificial base interfaces, ask yourself whether you really need to do things this way. For example, a more appropriate design might be:

Slice

```
interface Thing1
{
    // Operations here...
}

interface Thing2
{
    // Operations here...
}

interface ThingUser
{
    void putThing1(Thing1* thing);
    void putThing2(Thing2* thing);
}
```

With this design, `Thing1` and `Thing2` are not related, and `ThingUser` offers a separate operation for each type of proxy. The implementation of these operations does not need to use any down-casts, and all is well in our object-oriented world.

Another common use of empty base interfaces is the following:

Slice

```
interface PersistentObject {}

interface Thing1 extends PersistentObject
{
    // Operations here...
}

interface Thing2 extends PersistentObject
{
    // Operations here...
}
```

Clearly, the intent of this design is to place persistence functionality into the `PersistentObject` base *implementation* and require objects that want to have persistent state to inherit from `PersistentObject`. On the face of things, this is reasonable: after all, using inheritance in this way is a well-established design pattern, so what can possibly be wrong with it? As it turns out, there are a number of things that are wrong with this design:

- The above inheritance hierarchy is used to add *behavior* to `Thing1` and `Thing2`. However, in a strict OO model, behavior can be invoked only by sending messages. But, because `PersistentObject` has no operations, no messages can be sent.
  This raises the question of how the implementation of `PersistentObject` actually goes about doing its job; presumably, it knows something about the implementation (that is, the internal state) of `Thing1` and `Thing2`, so it can write that state into a database. But, if so, `PersistentObject`, `Thing1`, and `Thing2` can no longer be implemented in different address spaces because, in that case, `PersistentObject` can no longer get at the state of `Thing1` and `Thing2`.
  Alternatively, `Thing1` and `Thing2` use some functionality provided by `PersistentObject` in order to make their internal state persistent. But `PersistentObject` does not have any operations, so how would `Thing1` and `Thing2` actually go about achieving this? Again, the only way that can work is if `PersistentObject`, `Thing1`, and `Thing2` are implemented in a single address space and share implementation state behind the scenes, meaning that they cannot be implemented in different address spaces.

- The above inheritance hierarchy splits the world into two halves, one containing persistent objects and one containing non-persistent ones. This has far-reaching ramifications:
  - Suppose you have an existing application with already implemented, non-persistent objects. Requirements change over time and you find that you now would like to make some of your objects persistent. With the above design, you cannot do this unless you change the type of your objects because they now must inherit from `PersistentObject`. Of course, this is extremely bad news: not only do you have to change the implementation of your objects in the server, you also need to locate and update all the clients that are currently using your objects because they suddenly have a completely new type. What is worse, there is no way to keep things backward compatible: either all clients change with the server, or none of them do. It is impossible for some clients to remain "unupgraded".
  - The design does not scale to multiple features. Imagine that we have a number of additional behaviors that objects can inherit, such as serialization, fault-tolerance, persistence, and the ability to be searched by a search engine. We quickly end up in a mess of multiple inheritance. What is worse, each possible combination of features creates a completely separate type hierarchy. This means that you can no longer write operations that generically operate on a number of object types. For example, you cannot pass a persistent object to something that expects a non-persistent object, *even if the receiver of the object does not care about the persistence aspects of the object*. This quickly leads to fragmented and hard-to-maintain type systems. Before long, you will either find yourself rewriting your application or end up with something that is both difficult to use and difficult to maintain.

The foregoing discussion will hopefully serve as a warning: Slice is an *interface* definition language that has nothing to do with *implementation* (but empty interfaces almost always indicate that implementation state is shared via mechanisms other than defined interfaces). If you find yourself writing an empty interface definition, at least step back and think about the problem at hand; there may be a more appropriate design that expresses your intent more cleanly. If you do decide to go ahead with an empty interface regardless, be aware that, almost certainly, you will lose the ability to later change the distribution of the object model over physical server processes because you cannot place an address space boundary between interfaces that share hidden state.

Interface Versus Implementation Inheritance

Keep in mind that Slice interface inheritance applies only to *interfaces*. In particular, if two interfaces are in an inheritance relationship, this in no way implies that the implementations of those interfaces must also inherit from each other. You can choose to use implementation inheritance when you implement your interfaces, but you can also make the implementations independent of each other. (To C++ programmers, this often comes as a surprise because C++ uses implementation inheritance by default, and interface inheritance requires extra effort to implement.)

In summary, Slice inheritance simply establishes type compatibility. It says nothing about how interfaces are implemented and, therefore, keeps implementation choices open to whatever is most appropriate for your application.

See Also

- Interfaces, Operations, and Exceptions
- Operations
- User Exceptions
- Run-Time Exceptions
- Proxies for Ice Objects
- IceGrid

Classes

In addition to interfaces, Slice permits the definition of classes. Classes are like structures on steroids, with inheritance and the ability to hold optional data members.

Classes support inheritance and are therefore polymorphic: at run time, you can pass a class instance to an operation as long as the actual class type is derived from the formal parameter type in the operation's signature.

Topics

Simple Classes

A Slice class definition is similar to a structure definition, but uses the `class` keyword. For example:

Slice

```
module M
{
    class TimeOfDay
    {
        short hour;         // 0 - 23
        short minute;       // 0 - 59
        short second;       // 0 - 59
    }
}
```

Apart from the keyword `class`, this definition is identical to the structure example. You can use a Slice class wherever you can use a Slice structure (but, as we will see shortly, for performance reasons, you should not use a class where a structure is sufficient). Unlike structures, classes can be empty:

Slice

```
class EmptyClass {}    // OK
struct EmptyStruct {}  // Error
```

Much the same design considerations as for empty interfaces apply to empty classes: you should at least stop and rethink your approach before committing yourself to an empty class.

A class can define any number of data members, including optional data members. You can also specify a default value for a data member if its type is one of the following:

- An integral type (`byte`, `short`, `int`, `long`)
- A floating point type (`float` or `double`)
- `string`
- `bool`
- `enum`

For example:

Slice

```
class Location
{
    string name;
    Point pt;
    bool display = true;
    string source = "GPS";
}
```

The legal syntax for literal values is the same as for Slice constants, and you may also use a constant as a default value. The language mapping guarantees that data members are initialized to their declared default values using a language-specific mechanism.

See Also

- Structures
- Constants and Literals
- Optional Data Members

Class Inheritance

On this page:

Simple Inheritance

Unlike structures, classes support inheritance. For example:

Slice

```
module M
{
    class TimeOfDay
    {
        short hour;         // 0 - 23
        short minute;       // 0 - 59
        short second;       // 0 - 59
    }

    class DateTime extends TimeOfDay
    {
        short day;          // 1 - 31
        short month;        // 1 - 12
        short year;         // 1753 onwards
    }
}
```

This example illustrates one major reason for using a class: a class can be extended by inheritance, whereas a structure is not extensible. The previous example defines `DateTime` to extend the `TimeOfDay` class with a date.

> If you are puzzled by the comment about the year 1753, search the Web for "1752 date change". The intricacies of calendars for various countries prior to that year can keep you occupied for months...

Classes only support single inheritance. The following is illegal:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}

class Date
{
    short day;
    short month;
    short year;
}

class DateTime extends TimeOfDay, Date   // Error!
{
    // ...
}
```

A derived class also cannot redefine a data member of its base class:

Slice

```
class Base
{
    int integer;
}

class Derived extends Base
{
    int integer;                // Error, integer redefined
}
```

Implicit Inheritance from Value

All classes implicitly inherit from `Value`. This way, a `Value` parameter in an operation accepts any class instance.

See Also

- Structures

Class Inheritance Semantics

Classes use the same pass-by-value semantics as structures. If you pass a class instance to an operation, the class and all its members are passed. The usual type compatibility rules apply: you can pass a derived instance where a base instance is expected. If the receiver has static type knowledge of the actual derived run-time type, it receives the derived instance; otherwise, if the receiver does not have static type knowledge of the derived type, depending on the format used to encode the class, it will either fail to read the instance or slice the instance to the base type.

For an example, suppose we have the following definitions:

Slice

```
// In file Clock.ice:

module M
{
    class TimeOfDay
    {
        short hour;         // 0 - 23
        short minute;       // 0 - 59
        short second;       // 0 - 59
    }

    interface Clock
    {
        TimeOfDay getTime();
        void setTime(TimeOfDay time);
    }
}

// In file DateTime.ice:

#include <Clock.ice>

module M
{
    class DateTime extends TimeOfDay
    {
        short day;          // 1 - 31
        short month;        // 1 - 12
        short year;         // 1753 onwards
    }
}
```

Because `DateTime` is a sub-class of `TimeOfDay`, the server can return a `DateTime` instance from `getTime`, and the client can pass a `DateTime` instance to `setTime`. In this case, if both client and server are linked to include the code generated for both `Clock.ice` and `DateTime.ice`, they each receive the actual derived `DateTime` instance, that is, the actual run-time type of the instance is preserved.

Contrast this with the case where the server is linked to include the code generated for both `Clock.ice` and `DateTime.ice`, but the client is linked only with the code generated for `Clock.ice`. In other words, the server understands the type `DateTime` and can return a `DateTime` instance from `getTime`, but the client only understands `TimeOfDay`. In this case, there are two possible outcomes depending on the format used by the server to encode the instance:

- with the sliced format, the derived `DateTime` instance returned by the server is sliced to its `TimeOfDay` base type in the client

- with the compact format, `getTime` fails with the `Ice::NoObjectFactoryException` exception

> See Design Considerations for Objects for additional information on the sliced and compact formats.

Class hierarchies are useful if you need polymorphic *values* (instead of polymorphic *interfaces*). For example:

Slice

```
class Shape
{
    // Definitions for shapes, such as size, center, etc.
}

class Circle extends Shape
{
    // Definitions for circles, such as radius...
}

class Rectangle extends Shape
{
    // Definitions for rectangles, such as width and length...
}

sequence<Shape> ShapeSeq;

interface ShapeProcessor
{
    void processShapes(ShapeSeq ss);
}
```

Note the definition of `ShapeSeq` and its use as a parameter to the `processShapes` operation: the class hierarchy allows us to pass a polymorphic sequence of shapes (instead of having to define a separate operation for each type of shape).

The receiver of a `ShapeSeq` can iterate over the elements of the sequence and down-cast each element to its actual run-time type. (The receiver can also ask each element for its type ID to determine its type.)

See Also

- Structures
- Type IDs

Classes as Unions

Slice does not offer a dedicated union construct because it is redundant. By deriving classes from a common base class, you can create the same effect as with a union:

Slice

```
module M
{
    class Shape
    {
        // Definitions for shapes, such as size, center, etc.
    }

    class Circle extends Shape
    {
        // Definitions for circles, such as radius...
    }

    class Rectangle extends Shape
    {
        // Definitions for rectangles, such as width and length...
    }

    interface ShapeShifter
    {
        Shape translate(Shape s, long xDistance, long yDistance);
    }
}
```

The parameter `s` of the `translate` operation can be viewed as a union of two members: a `Circle` and a `Rectangle`. The receiver of a `Shape` instance can use the type ID of the instance to decide whether it received a `Circle` or a `Rectangle`. Alternatively, if you want something more along the lines of a conventional discriminated union, you can use the following approach:

Slice

```
class UnionDiscriminator
{
    int d;
}

class Member1 extends UnionDiscriminator
{
    // d == 1
    string s;
    float f;
}

class Member2 extends UnionDiscriminator
{
    // d == 2
    byte b;
    int i;
}
```

With this approach, the `UnionDiscriminator` class provides a discriminator value. The "members" of the union are the classes that are derived from `UnionDiscriminator`. For each derived class, the discriminator takes on a distinct value. The receiver of such a union uses the discriminator value in a `switch` statement to select the active union member.

See Also

- Type IDs

Self-Referential Classes

Classes can be self-referential.

For example:

Slice

```
class Link
{
    SomeType value;
    Link next;
}
```

This looks very similar to the self-referential interface example, but the semantics are very different. Note that `value` and `next` are data members, not operations, and that the type of `next` is `Link` (*not* `Link*`). As you would expect, this forms the same linked list arrangement as the `Link` interface in Self-Referential Interfaces: each instance of a `Link` class contains a `next` member that points at the next link in the chain; the final link's `next` member contains a null value. So, what looks like a class including itself really expresses pointer semantics: the `next` data member contains a pointer to the next link in the chain.

You may be wondering at this point what the difference is then between the `Link` interface in Self-Referential Interfaces and the `Link` class shown above. The difference is that classes have *value* semantics, whereas proxies have *reference* semantics. To illustrate this, consider the `Link` *interface* from Self-Referential Interfaces once more:

Slice

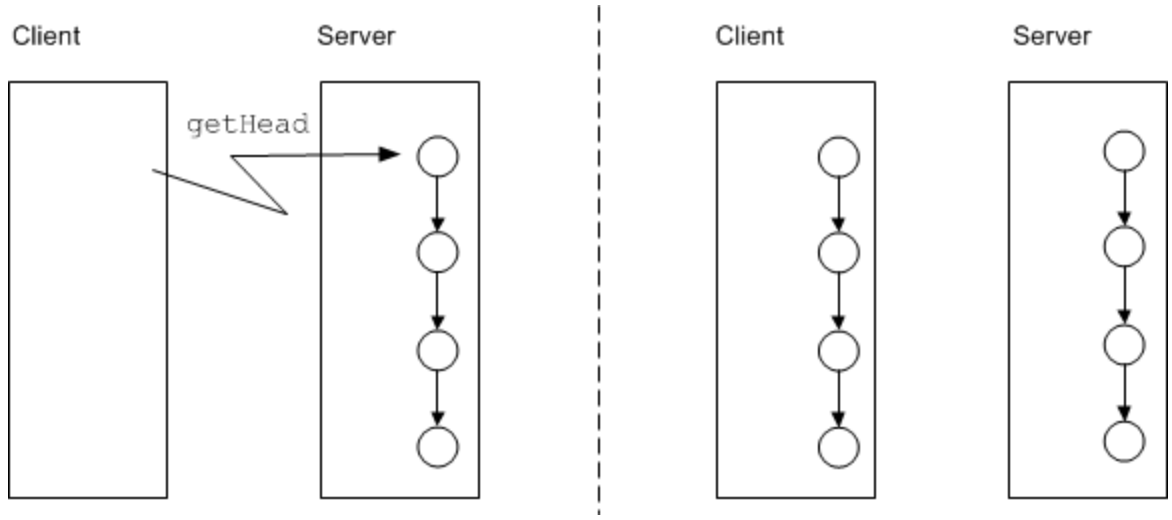
```
interface Link
{
    idempotent SomeType getValue();
    idempotent Link*    next();
}
```

Here, `getValue` and `next` are both operations and the return value of `next` is `Link*`, that is, next returns a *proxy*. A proxy has *reference* semantics, that is, it denotes an object somewhere. If you invoke the `getValue` operation on a `Link` proxy, a message is sent to the (possibly remote) servant for that proxy. In other words, for proxies, the object stays put in its server process and we access the state of the object via remote procedure calls. Compare this with the definition of our `Link` *class*:

Slice

```
class Link
{
    SomeType value;
    Link next;
}
```

Here, `value` and `next` are data members and the type of next is `Link`, which has *value* semantics. In particular, while `next` looks and feels like a pointer, *it cannot denote an instance in a different address space*. This means that if we have a chain of `Link` instances, all of the instances are in our local address space and, when we read or write a value data member, we are performing local address space operations. This means that an operation that returns a `Link` instance, such as `getHead`, does not just return the head of the chain, *but the entire chain*, as shown:



Class version of `Link` before and after calling `getHead`.

On the other hand, for the interface version of `Link`, we do not know where all the links are physically implemented. For example, a chain of four links could have each object instance in its own physical server process; those server processes could be each in a different continent. If you have a proxy to the head of this four-link chain and traverse the chain by invoking the `next` operation on each link, you will be sending four remote procedure calls, one to each object.

Self-referential classes are particularly useful to model graphs. For example, we can create a simple expression tree along the following lines:

Slice

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not }
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or }

class Node {}

class UnaryOperator extends Node
{
    UnaryOp operator;
    Node operand;
}

class BinaryOperator extends Node
{
    BinaryOp op;
    Node operand1;
    Node operand2;
}

class Operand extends Node
{
    long val;
}
```

The expression tree consists of leaf nodes of type `Operand`, and interior nodes of type `UnaryOperator` and `BinaryOperator`, with one or two descendants, respectively. All three of these classes are derived from a common base class `Node`. Note that `Node` is an empty class.

This is one of the few cases where an empty base class is justified. (See the discussion on empty interfaces; once we add operations to this class hierarchy, the base class is no longer empty.)

If we write an operation that, for example, accepts a `Node` parameter, passing that parameter results in transmission of the entire tree to the server:

Slice

```
interface Evaluator
{
    long eval(Node expression); // Send entire tree for evaluation
}
```

Self-referential classes are not limited to acyclic graphs; the Ice run time permits loops: it ensures that no resources are leaked and that infinite loops are avoided during marshaling.

See Also

- Classes with Operations
- Self-Referential Interfaces

Classes Versus Structures

One obvious question to ask is: why does Ice provide structures as well as classes, when classes obviously can be used to model structures? The answer has to do with the cost of implementation: classes provide a number of features that are absent for structures:

- Classes support inheritance.
- Classes can be self-referential.
- Classes can have optional data members.

Obviously, an implementation cost is associated with the additional features of classes, both in terms of the size of the generated code and the amount of memory and CPU cycles consumed at run time. On the other hand, structures are simple collections of values ("plain old structs") and are implemented using very efficient mechanisms. This means that, if you use structures, you can expect better performance and smaller memory footprint than if you would use classes. Use a class only if you need at least one of its more powerful features.

See Also

- Structures
- Classes with Operations
- Classes Implementing Interfaces

Classes with Operations

Classes, in addition to data members, can have operations.

> **Deprecated Feature**
> Operations on classes are deprecated as of Ice 3.7. Skip this page unless you need to communicate with old applications that rely on this feature.

The syntax for operation definitions in classes is identical to the syntax for operations in interfaces. For example, we can modify the expression tree from Self-Referential Classes as follows:

Slice

```
module M
{
    enum UnaryOp { UnaryPlus, UnaryMinus, Not }
    enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or }

    class Node
    {
        idempotent long eval();
    }

    class UnaryOperator extends Node
    {
        UnaryOp operator;
        Node operand;
    }

    class BinaryOperator extends Node
    {
        BinaryOp op;
        Node operand1;
        Node operand2;
    }

    class Operand
    {
        long val;
    }
}
```

The only change compared to the version in Self-Referential Classes is that the `Node` class now has an `eval` operation. The semantics of this are as for a virtual member function in C++: each derived class inherits the operation from its base class and can choose to override the operation's definition. For our expression tree, the `Operand` class provides an implementation that simply returns the value of its `val` member, and the `UnaryOperator` and `BinaryOperator` classes provide implementations that compute the value of their respective subtrees. If we call `eval` on the root node of an expression tree, it returns the value of that tree, regardless of whether we have a complex expression or a tree that consists of only a single `Operand` node.

Operations on classes are normally executed in the caller's address space, that is, operations on classes are *local* operations that do not result in a remote procedure call.

It is also possible to invoke an operation on a remote class instance.

Of course, this immediately raises an interesting question: what happens if a client receives a class instance with operations from a server, but client and server are implemented in different languages? Classes with operations require the receiver to supply a factory for instances of the class. The Ice run time only marshals the data members of the class. If a class has operations, the receiver of the class must provide a class factory that can instantiate the class in the receiver's address space, and the receiver is responsible for providing an implementation of the class's operations.

Therefore, if you use classes with operations, it is understood that client and server each have access to an implementation of the class's operations. No code is shipped over the wire (which, in an environment of heterogeneous nodes using different operating systems and languages is infeasible).

See Also

- Self-Referential Classes
- Pass-by-Value Versus Pass-by-Reference

Classes Implementing Interfaces

Deprecated Feature
Like operations on classes, classes inheriting from interfaces or providing remote operations like interfaces are deprecated as of Ice 3.7. Skip this page unless you need to communicate with old applications that rely on this feature.

A Slice class can also be used as a servant in a server, that is, an instance of a class can be used to provide the behavior for an interface, for example:

Slice

```
module M
{
    interface Time
    {
        idempotent TimeOfDay getTime();
        idempotent void setTime(TimeOfDay time);
    }

    class Clock implements Time
    {
        TimeOfDay time;
    }
}
```

The `implements` keyword indicates that the class `Clock` provides an *implementation* of the `Time` interface. The class can provide data members and operations of its own; in the preceding example, the `Clock` class stores the current time that is accessed via the `Time` interface. A class can implement several interfaces, for example:

Slice

```
interface Time
{
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
}

interface Radio
{
    idempotent void setFrequency(long hertz);
    idempotent void setVolume(long dB);
}

class RadioClock implements Time, Radio
{
    TimeOfDay time;
    long hertz;
}
```

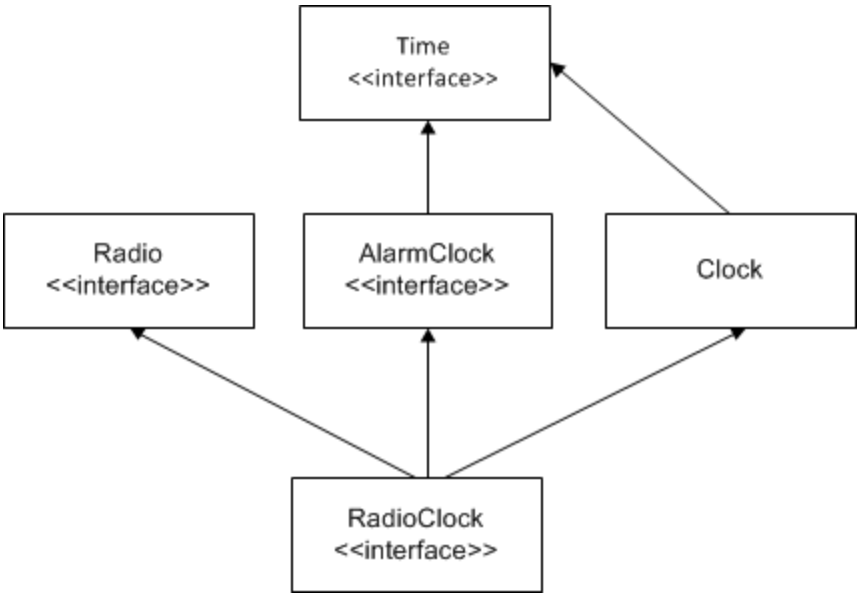
The class `RadioClock` implements both `Time` and `Radio` interfaces.

A class, in addition to implementing an interface, can also extend another class:

Slice

```
interface Time
{
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
}

class Clock implements Time
{
    TimeOfDay time;
}

interface AlarmClock extends Time
{
    idempotent TimeOfDay getAlarmTime();
    idempotent void setAlarmTime(TimeOfDay alarmTime);
}

interface Radio
{
    idempotent void setFrequency(long hertz);
    idempotent void setVolume(long dB);
}

class RadioAlarmClock extends Clock
                      implements AlarmClock, Radio
{
    TimeOfDay alarmTime;
    long hertz;
}
```

These definitions result in the following inheritance graph:



A Class using implementation and interface inheritance.

For this definition, `Radio` and `AlarmClock` are abstract interfaces, and `Clock` and `RadioAlarmClock` are concrete classes. As for Java, a class can implement multiple interfaces, but can extend at most one class.

See Also

- Class Inheritance Limitations

Class Inheritance Limitations

As for interface inheritance, a class cannot redefine an operation or data member that it inherits from a base interface or class. For example:

Slice

```
module M
{
    interface BaseInterface
    {
        void op();
    }

    class BaseClass
    {
        int member;
    }

    class DerivedClass extends BaseClass implements BaseInterface
    {
        void someOperation();       // OK
        int op();                   // Error!
        int  someMember;            // OK
        long member;                // Error!
    }
}
```

See Also

- Interface Inheritance

Pass-by-Value Versus Pass-by-Reference

On this page:

Passing a Class by Reference

As we saw in Self-Referential Classes, classes naturally support pass-by-value semantics: passing a class transmits the data members of the class to the receiver. Any changes made to these data members by the receiver affect only the receiver's copy of the class; the data members of the sender's class are not affected by the changes made by the receiver.

In addition to passing a class by value, you can pass a class by reference.

> **Deprecated Feature**
> Passing class instances by reference - as proxies - is deprecated as of Ice 3.7. Skip this paragraph unless you need to communicate with old applications that rely on this feature.

For example:

Slice

```
module M
{
    class TimeOfDay
    {
        short hour;
        short minute;
        short second;
        string format();
    }

    interface Example
    {
        TimeOfDay* get();  // Note: returns a proxy!
    }
}
```

Note that the `get` operation returns a *proxy* to a `TimeOfDay` class and not a `TimeOfDay` instance itself. The semantics of this are as follows:

- When the client receives a `TimeOfDay` proxy from the `get` call, it holds a proxy that differs in no way from an ordinary proxy for an interface.
- The client can invoke operations via the proxy, but *cannot* access the data members. This is because proxies do not have the concept of data members, but represent interfaces: even though the `TimeOfDay` class has data members, only its *operations* can be accessed via a the proxy.

The net effect is that, in the preceding example, the server holds an instance of the `TimeOfDay` class. A proxy for that instance was passed to the client. The only thing the client can do with this proxy is to invoke the `format` operation. The implementation of that operation is provided by the server and, when the client invokes `format`, it sends an RPC message to the server just as it does when it invokes an operation on an interface. The implementation of the `format` operation is entirely up to the server. (Presumably, the server will use the data members of the `TimeOfDay` instance it holds to return a string containing the time to the client.)

The preceding example looks somewhat contrived for classes only. However, it makes perfect sense if classes implement interfaces: parts of your application can exchange class instances (and, therefore, state) by value, whereas other parts of the system can treat these instances as remote interfaces.

For example:

Slice

```
interface Time
{
    string format();
    // ...
}

class TimeOfDay implements Time
{
    short hour;
    short minute;
    short second;
}

interface I1
{
     TimeOfDay get();           // Pass by value
     void put(TimeOfDay time);  // Pass by value
}

interface I2
{
    Time* get();                // Pass by reference
}
```

In this example, clients dealing with interface `I1` are aware of the `TimeOfDay` class and pass it by value whereas clients dealing with interface `I2` deal only with the `Time` interface. However, the actual implementation of the `Time` interface in the server uses `TimeOfDay` inst ances.

Be careful when designing systems that use such mixed pass-by-value and pass-by-reference semantics. Unless you are clear about what parts of the system deal with the interface (pass by reference) aspects and the class (pass by value) aspects, you can end up with something that is more confusing than helpful.

Object vs Value

In non-local operations, you can use an `Object` or `Value` parameter to mean "accept any class or interface passed by value". `Object` and `Value` are synonymous in this context, and mean any value.

For example, interface `I1` in the preceding example could be rewritten as:

Slice

```
interface UntypedI1
{
     Value get();           // Pass by value
     void put(Value time);  // Pass by value
}
```

or

Slice

```
interface UntypedI1
{
     Object get();           // Pass by value
     void put(Object time);  // Pass by value
}
```

In local operations, `Object` and `Value` are not interchangeable: `Object` designates the base class for all servants, while `Value` designates the base class for all mapped classes.

Finally for proxies, `Object*` always means any proxy, while `Value*` is never correct.

See Also

- Self-Referential Classes

Passing Interfaces by Value

Deprecated Feature
Passing interfaces by value is deprecated as of Ice 3.7, just like passing classes by reference. Skip this page unless you need to communicate with old applications that rely on this feature.

Consider the following definitions:

Slice

```
module M
{
    interface Time
    {
        idempotent TimeOfDay getTime();
        // ...
    }

    interface Record
    {
        void addTimeStamp(Time t); // Note: Time t, not Time* t
        // ...
    }
}
```

Note that `addTimeStamp` accepts a parameter of type `Time`, not of type `Time*`. The question is, what does it mean to pass an interface *by value*? Obviously, at run time, we cannot pass an an actual interface to this operation because interfaces are abstract and cannot be instantiated. Neither can we pass a proxy to a `Time` object to `addTimeStamp` because a proxy cannot be passed where an interface is expected.

However, what we *can* pass to `addTimeStamp` is something that is not abstract and derives from the `Time` interface. For example, at run time, we could pass an instance of the `TimeOfDay` class we saw earlier. Because the `TimeOfDay` class derives from the `Time` interface, the class type is compatible with the formal parameter type `Time` and, at run time, what is sent over the wire to the server is the `TimeOfDay` class instance.

See Also

- Pass-by-Value Versus Pass-by-Reference

Classes with Compact Type IDs

You can optionally associate a numeric identifier with a class. The Ice run time substitutes this value, known as a *compact type ID*, in place of its equivalent string type ID during marshaling to conserve space. The compact type ID follows immediately after the class name, enclosed in parentheses:

Slice

```
module M
{
    class CompactExample(4)
    {
        // ...
    }
}
```

In this example, the Ice run time marshals the value 4 instead of its string equivalent `"::M::CompactExample"`. The specified value must be a non-negative integer that is unique within the translation unit.

> Using values less than 255 produces the most efficient encoding.

See Also

- Classes
- Type IDs
- Data Encoding for Class Type IDs

Value Factories

Prior to Ice 3.7, an application needed to register an object factory with the Ice run time for two use cases:

1. to successfully unmarshal an instance of a Slice class that defined operations, or
2. to supply a custom implementation of a Slice class, regardless of whether that class defined operations.

Since Ice 3.7 deprecates classes with operations, we now refer to instances of Slice classes as *values* and the Ice run time provides a new (but similar) API for managing value factories. Generally speaking, applications will rarely need to use this API, with use case #2 above now being the primary motivation.

The following Slice definitions comprise the value factory API:

Slice

```
module Ice
{
    local interface ValueFactory
    {
        Value create(string type);
    }

    local interface ValueFactoryManager
    {
        void add(ValueFactory factory, string type);
        ValueFactory find(string type);
    }

    local interface Communicator
    {
        ValueFactoryManager getValueFactoryManager();
        // ...
    }
}
```

An application-defined value factory must provide an implementation of the `ValueFactory` interface. Its `create` operation receives the Slice type ID corresponding to the Slice class that the Ice run time is attempting to unmarshal. The `create` implementation can return nil if it's unable to instantiate the type or doesn't recognize the type, otherwise the factory must return an instance of the requested type or a type derived from the requested type.

The Ice run time supplies a default implementation of the `ValueFactoryManager` interface, although an application can optionally substitute its own implementation during communicator initialization. You can obtain the value factory manager by calling `getValueFactoryManager` on the communicator object. The manager's `add` operation registers a factory for a particular Slice type ID, or you can pass an empty string as the type and Ice will use that factory as the default in cases where no other factory was registered for a type. The `add` operation raises `AlreadyRegisteredException` if another factory has already been registered for the specified type.

Finally, the manager's `find` operation returns the factory registered for a type, or nil if no match was found.

Please refer to the relevant language mapping chapters for instructions on using the value factory API in your programming language.

See Also

- Classes
- Type IDs

Forward Declarations

Both interfaces and classes can be forward declared. Forward declarations permit the creation of mutually dependent objects, for example:

Slice

```
module Family
{
    interface Child;            // Forward declaration

    sequence<Child*> Children;  // OK

    interface Parent
    {
        Children getChildren(); // OK
    }

    interface Child           // Definition
    {
        Parent* getMother();
        Parent* getFather();
    }
}
```

Without the forward declaration of `Child`, the definition obviously could not compile because `Child` and `Parent` are mutually dependent interfaces. You can use forward-declared interfaces and classes to define types (such as the `Children` sequence in the previous example). Forward-declared interfaces and classes are also legal as the type of a structure, exception, or class member, as the value type of a dictionary, and as the parameter and return type of an operation.

The definition of a forward-declared interface or class must appear in the same translation unit if that type is used as a proxy, or if that type is used in any context in which it could be marshaled:

Slice

```
module Family
{
    interface Child;            // Forward declaration
    class Chore;                // Forward declaration

    sequence<Child*> Children;  // Error - undefined proxy type!

    interface Parent
    {
        Chore nextChore();      // Error - undefined class type!
    }
}
```

Finally, you cannot inherit from a forward-declared interface or class until after its definition has been seen by the compiler:

Slice

```
interface Base;                          // Forward declaration

interface Derived1 extends Base {}       // Error!

interface Base {}                        // Definition

interface Derived2 extends Base {}       // OK, definition was seen
```

Not inheriting from a forward-declared base interface or class until its definition is seen is necessary because, otherwise, the compiler could not enforce that derived interfaces must not redefine operations that appear in base interfaces.

A multi-pass compiler could be used, but the added complexity is not worth it.

See Also

- Interfaces, Operations, and Exceptions
- Classes

Optional Data Members

On this page:

Overview of Optional Data Members

A data member of a Slice class or exception may be declared as optional to indicate that a program can leave its value unset. Data members not declared as optional are known as *required* members; a program must supply legal values for all required members.

Declaring Optional Data Members

Each optional data member in a type must be assigned a unique, non-negative integer *tag*:

Slice

```
module M
{
    class C
    {
        string name;
        bool active;
        optional(2) string alternateName;
        optional(5) int overrideCode;
    }
}
```

It is legal for a base type's tag to be reused by a derived type:

Slice

```
exception Base
{
    optional(1) int systemCode;
}

exception Derived extends Base
{
    optional(1) string diagnostic; // OK
}
```

The scope of a tag is limited to its enclosing type and has no effect on base or derived types.

Language mappings specify an API for setting an optional member and testing whether a member is set. Here is an example in C++:

C++11 C++98

```
auto c = make_shared<C>();
c->name = "xyz";           // required
c->active = true;          // required
c->alternateName = "abc";  // optional
c->overrideCode = 42;      // optional

if(c->alternateName)
{
    cout << "alt name = " << c->alternateName << endl;
}
```

```
CPtr c = new C;
c->name = "xyz";           // required
c->active = true;          // required
c->alternateName = "abc";  // optional
c->overrideCode = 42;      // optional

if(c->alternateName)
{
    cout << "alt name = " << c->alternateName << endl;
}
```

As you can see, the C++ language mapping makes setting an optional member as simple as assigning it a value. Refer to the language mapping sections for more details on the optional data member API.

> A well-behaved program must test for the presence of an optional member and not assume that it is always set. Dereferencing an unset optional member causes a run-time error.

In all supported language mappings, an optional data member's initial condition is unset if not otherwise assigned during construction. Again using C++ as an example:

C++11 C++98

```
auto c = make_shared<C>();           // default constructor
assert(!c->alternateName);           // not set

c = make_shared<C>("xyz", true, "abc", 42); // one-shot constructor
assert(c->alternateName);                   // set by constructor
```

```
CPtr c = new C;           // default constructor
assert(!c->alternateName); // not set

c = new C("xyz", true, "abc", 42); // one-shot constructor
assert(c->alternateName);          // set by constructor
```

Optional Data Members with Default Values

You can declare a default value for optional members just as you can for required members:

Slice

```
class C
{
    string name;
    bool active = true;
    optional(2) string alternateName;
    optional(5) int overrideCode = -1;
}
```

An optional data member with a default value is considered to be set by default:

C++11 C++98

```
auto c = make_shared<C>();      // default constructor
assert(!c->alternateName);      // not set
assert(c->overrideCode);        // set to default value
```

```
CPtr c = new C;            // default constructor
assert(!c->alternateName); // not set
assert(c->overrideCode);   // set to default value
```

Each language mapping provides an API for resetting an optional data member to its unset condition.

See Also

- Classes
- User Exceptions
- Optional Values

Type IDs

Interface, class and user exception Slice types have an internal type identifier, known as the *type ID*. The type ID is simply the fully-qualified name of each type. For example, the type ID of the `Child` interface in the preceding example is `::Family::Children::Child`. A type ID starts with a leading `::` and is formed by starting with the global scope (`::`) and forming the fully-qualified name of a type by appending each module name in which the type is nested, and ending with the name of the type itself; the components of the type ID are separated by `::`.

The type ID of the Slice `Object` type is `::Ice::Object`.

Type IDs are used internally by the Ice run time as a unique identifier for each type. For example, when an exception is raised, the marshaled form of the exception that is returned to the client is preceded by its type ID on the wire. The client-side run time first reads the type ID and, based on that, unmarshals the remainder of the data as appropriate for the type of the exception.

Type IDs are also used by the `ice_isA` operation.

See Also

- ice_isA

Operations on Object

The `Object` interface has a number of operations. We cannot define type `Object` in Slice because `Object` is a keyword; regardless, here is what (part of) the definition of `Object` would look like if it were legal:

Slice

```
sequence<string> StrSeq;

interface Object                        // "Pseudo" Slice!
{
    idempotent void   ice_ping();
    idempotent bool   ice_isA(string typeID);
    idempotent string ice_id();
    idempotent StrSeq ice_ids();
    // ...
}
```

Note that, apart from the illegal use of the keyword `Object` as the interface name, the operation names all contain the `ice_` prefix. This prefix is reserved for use by Ice and cannot clash with a user-defined operation. This means that all Slice interfaces can inherit from `Object` without name clashes. We discuss these built-in operations below.

On this page:

ice_ping

All interfaces support the `ice_ping` operation. That operation is useful for debugging because it provides a basic reachability test for an object: if the object exists and a message can successfully be dispatched to the object, `ice_ping` simply returns without error. If the object cannot be reached or does not exist, `ice_ping` throws a run-time exception that provides the reason for the failure.

ice_isA

The `ice_isA` operation accepts a type identifier (such as the identifier returned by `ice_id`) and tests whether the target object supports the specified type, returning `true` if it does. You can use this operation to check whether a target object supports a particular type. For example, referring to the diagram Implicit Inheritance from Object once more, assume that you are holding a proxy to a target object of type `AlarmClock`. The table below illustrates the result of calling `ice_isA` on that proxy with various arguments. (We assume that all types in the Implicit inheritance from Object diagram are defined in a module `Times`):

Argument	Result
`::Ice::Object`	`true`
`::Times::Clock`	`true`
`::Times::AlarmClock`	`true`
`::Times::Radio`	`false`
`::Times::RadioClock`	`false`

Calling `ice_isA` on a proxy denoting an object of type AlarmClock.

As expected, `ice_isA` returns true for `::Times::Clock` and `::Times::AlarmClock` and also returns true for `::Ice::Object` (because all interfaces support that type). Obviously, an `AlarmClock` supports neither the `Radio` nor the `RadioClock` interfaces, so `ice_isA` returns false for these types.

ice_id

The `ice_id` operation returns the type ID of the most-derived type of an interface.

ice_ids

The `ice_ids` operation returns a sequence of type IDs that contains all of the type IDs supported by an interface. For example, for the RadioClock interface in Implicit inheritance from Object, `ice_ids` returns a sequence containing the type IDs `::Ice::Object`, `::Times::Clock`, `::Times::AlarmClock`, `::Times::Radio`, and `::Times::RadioClock`.

See Also

- Type IDs
- Interface Inheritance
- Implicit inheritance from Object

Local Types

In order to access certain features of the Ice run time, you must use APIs that are provided by libraries. However, instead of defining an API that is specific to each implementation language, Ice defines its APIs in Slice using the `local` keyword. The advantage of defining APIs in Slice is that a single definition suffices to define the API for all possible implementation languages. The actual language-specific API is then generated by the Slice compiler for each implementation language. Types that are provided by Ice libraries are defined using the Slice `local` keyword.

For example:

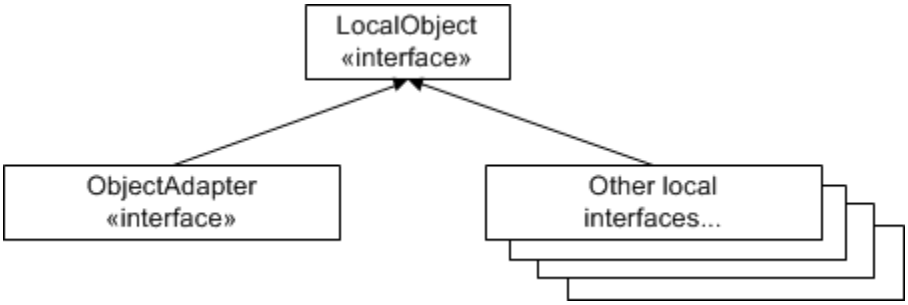
Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        // ...
    }
}
```

Any Slice definition (not just interfaces) can have a `local` modifier. If the `local` modifier is present, the Slice compiler does not generate marshaling code for the corresponding type. This means that a local type can *never* be accessed remotely because it cannot be transmitted between client and server. (The Slice compiler prevents use of `local` types in non-`local` contexts.)

> Slice `modules` and `constants` are inherently local and cannot be prefixed with the `local` modifier.

Local interfaces and local classes do *not* inherit from `Object` resp. `Value`. Instead, local interfaces and classes have their own, completely separate inheritance hierarchy. At the root of this hierarchy is the type `LocalObject`, as shown:



Inheritance from `LocalObject`.

Because local interfaces and local classes form a completely separate inheritance hierarchy, you cannot pass a local interface where a non-local interface is expected, and vice-versa.

You rarely need to define local types for your own applications — the `local` keyword exists mainly to allow definition of APIs for the Ice run time. (Because local objects cannot be invoked remotely, there is little point for an application to define local objects; it might as well define ordinary programming-language objects instead.) However, there is one exception to this rule: servant locators must be implemented as local objects.

See Also

- Servant Locators

Names and Scoping

Slice has a number of rules regarding identifiers. You will typically not have to concern yourself with these. However, occasionally, it is good to know how Slice uses naming scopes and resolves identifiers.

On this page:

Naming Scope

The following Slice constructs establish a naming scope:

- the global (file) scope
- modules
- interfaces
- classes
- structures
- exceptions
- parameter lists

Within a naming scope, identifiers must be unique, that is, you cannot use the same identifier for different purposes. For example:

Slice

```
interface Bad
{
    void op(int p, string p);   // Error!
}
```

Because a parameter list forms a naming scope, it is illegal to use the same identifier p for different parameters. Similarly, data members, operation names, interface and class names, etc. must be unique within their enclosing scope.

Case Sensitivity

Identifiers that differ only in case are considered identical, so you must use identifiers that differ not only in capitalization within a naming scope. For example:

Slice

```
struct Bad
{
    int    m;
    string M;   // Error!
}
```

The Slice compiler also enforces consistent capitalization for identifiers. Once you have defined an identifier, you must use the same capitalization for that identifier thereafter. For example, the following is in error:

Slice

```
sequence<string> StringSeq;

interface Bad
{
    stringSeq op();     // Error!
}
```

Note that identifiers must not differ from a Slice keyword in case only. For example, the following is in error:

Slice

```
interface Module      // Error, "module" is a keyword
{
    // ...
}
```

Qualified Names

The scope-qualification operator `::` allows you to refer to a type in a non-local scope. For example:

Slice

```
module Types
{
    sequence<long> LongSeq;
}

module MyApp
{
    sequence<Types::LongSeq> NumberTree;
}
```

Here, the qualified name `Types::LongSeq` refers to `LongSeq` defined in module `Types`. The global scope is denoted by a leading `::`, so we could also refer to `LongSeq` as `::Types::LongSeq`.

The scope-qualification operator also allows you to create mutually dependent interfaces that are defined in different modules. The obvious attempt to do this fails:

Slice

```
module Parents
{
    interface Children::Child;  // Syntax error!
    interface Mother
    {
        Children::Child* getChild();
    }
    interface Father
    {
        Children::Child* getChild();
    }
}

module Children
{
    interface Child
    {
        Parents::Mother* getMother();
        Parents::Father* getFather();
    }
}
```

This fails because it is syntactically illegal to forward-declare an interface in a different module. To make it work, we must use a reopened module:

Slice

```
module Children
{
    interface Child;                    // Forward declaration
}

module Parents
{
    interface Mother
    {
        Children::Child* getChild();    // OK
    }
    interface Father
    {
        Children::Child* getChild();    // OK
    }
}

module Children                         // Reopen module
{
    interface Child                     // Define Child
    {
        Parents::Mother* getMother();
        Parents::Father* getFather();
    }
}
```

While this technique works, it is probably of dubious value: mutually dependent interfaces are, by definition, tightly coupled. On the other hand, modules are meant to be used to place related definitions into the same module, and unrelated definitions into different modules. Of course, this begs the question: if the interfaces are so closely related that they depend on each other, why are they defined in different modules? In the interest of clarity, you probably should avoid this construct, even though it is legal.

Names in Nested Scopes

Names defined in an enclosing scope can be redefined in an inner scope. For example, the following is legal:

Slice

```
module Outer
{
    sequence<string> Seq;

    module Inner
    {
        sequence<short> Seq;
    }
}
```

Within module `Inner`, the name `Seq` refers to a sequence of `short` values and hides the definition of `Outer::Seq`. You can still refer to the other definition by using explicit scope qualification, for example:

Slice

```
module Outer
{
    sequence<string> Seq;

    module Inner
    {
        sequence<short> Seq;

        struct Confusing
        {
            Seq          a;     // Sequence of short
            ::Outer::Seq b;     // Sequence of string
        }
    }
}
```

Needless to say, you should try to avoid such redefinitions — they make it harder for the reader to follow the meaning of a specification.

Same-named constructs cannot be nested inside each other in certain situations. For example, a module named `M` cannot (recursively) contain any construct also named `M`. The same is true for interfaces and classes, which cannot define an operation with the same name as the enclosing interface or class. For example, the following examples are all in error:

Slice

```
module M
{
    interface M { /* ... */ }  // Error!

    interface I
    {
        void I();               // Error!
    }
}

module Outer
{
    module Inner
    {
        interface outer       // Error, even if case differs!
        {
            // ...
        }
    }
}
```

The reason for this restriction is that nested types that have the same name are difficult to map into some languages. For example, C++ and

Java reserve the name of a class as the name of the constructor, so an interface `I` could not contain an operation named `I` without artificial rules to avoid the name clash.

Similarly, some languages (such as C# prior to version 2.0) do not permit a qualified name to be anchored at the global scope. If a nested module or type is permitted to have the same name as the name of an enclosing module, it can become impossible to generate legal code in some cases.

In the interest of simplicity, Slice prohibits the name of a nested module or type from being the same as the name of one of its enclosing modules.

Introduced Identifiers

Within a naming scope, an identifier is introduced at the point of first use; thereafter, within that naming scope, the identifier cannot change meaning.

For example:

Slice

```
module M
{
    sequence<string> Seq;

    interface Bad
    {
        Seq op1();      // Seq and op1 introduced here
        int Seq();      // Error, Seq has changed meaning
    }
}
```

The declaration of `op1` uses `Seq` as its return type, thereby introducing `Seq` into the scope of interface `Bad`. Thereafter, `Seq` can only be used as a type name that denotes a sequence of strings, so the compiler flags the declaration of the second operation as an error.

Note that fully-qualified identifiers are not introduced into the current scope:

Slice

```
module M
{
    sequence<string> Seq;

    interface Bad
    {
        ::M::Seq op1(); // Only op1 introduced here
        int Seq();      // OK
    }
}
```

In general, a fully-qualified name (one that is anchored at the global scope and, therefore, begins with a `::` scope resolution operator) does not introduce any name into the current scope. On the other hand, a qualified name that is not anchored at the global scope introduces only the first component of the name:

Slice

```
module M
{
    sequence<string> Seq;

    interface Bad
    {
        M::Seq op1();   // M and op1 introduced here, but not Seq
        int Seq();      // OK
    }
}
```

Name Lookup Rules

When searching for the definition of a name that is not anchored at the global scope, the compiler first searches backward in the current scope of a definition of the name. If it can find the name in the current scope, it uses that definition. Otherwise, the compiler successively searches enclosing scopes for the name until it reaches the global scope. Here is an example to illustrate this:

Slice

```
module M1
{
    sequence<double> Seq;

    module M2
    {
        sequence<string> Seq;   // OK, hides ::M1::Seq

        interface Base
        {
            Seq op1();          // Returns sequence of string
        }
    }

    module M3
    {
        interface Derived extends M2::Base
        {
            Seq op2();          // Returns sequence of double
        }

        sequence<bool> Seq;     // OK, hides ::M1::Seq

        interface I
        {
            Seq op();           // Returns sequence of bool
        }
    }

    interface I
    {
        Seq op();               // Returns sequence of double
    }
}
```

Note that `M2::Derived::op2` returns a sequence of `double`, even though `M1::Base::op1` returns a sequence of `string`. That is, the meaning of a type in a base interface is irrelevant to determining its meaning in a derived interface — the compiler always searches for a definition only in the current scope and enclosing scopes, and never takes the meaning of a name from a base interface or class.

Scoping Rules for Parameters and Data Members

A Slice operation creates a new naming scope in which all parameter names must be unique:

Slice

```
interface I
{
    void op1(string p, int P);    // Error, differs only in case
    void op2(int n, out int n);   // Error, duplicate
    void op3(string s, int i);    // OK
}
```

It's legal for parameters to reuse the names of symbols in enclosing scopes, including the name of the operation, class, interface or module:

Slice

```
module M
{
    sequence<string> Seq;

    interface I
    {
        string query(string query);   // OK to reuse operation name
        void op1(int I);              // OK to reuse name of enclosing
type
        void op2(Seq Seq);            // OK to reuse type name
        void op3(int M);              // OK to reuse module name
    }
}
```

The rules for data members are similar to those of parameters:

Structures	Member names must be unique within the structure.
Exceptions	Member names must be unique within the exception, including any members inherited from base exceptions.
Classes	Member names must be unique within the class, including any members inherited from base classes. Members must not duplicate the names of operations defined by the class or inherited by any base classes or interfaces.

As for parameters, data members can reuse the names of symbols in enclosing scopes. The examples below illustrate these rules:

Slice

```
module M
{
    struct S
    {
        int i;
        string s;    // OK to reuse name of enclosing type
        long I;      // Error, differs only in case
        bool M;      // OK to reuse module name
    }

    interface I
    {
        void op();
    }

    class Base
    {
        string name;
    }

    class C extends Base implements I
    {
        S S;         // OK to reuse type name
        byte c;      // OK to reuse name of enclosing type
        string op;   // Error, duplicates inherited I::op
        string Name; // Error, differs only in case from Base::name
    }

    exception ErrorBase
    {
        string reason;
    }

    exception Error extends ErrorBase
    {
        long error;  // OK to reuse name of enclosing type
        int reason;  // Error, duplicates inherited ErrorBase::reason
    }
}
```

Scoping Rules in Prior Ice Releases

The scoping rules for parameters and data members were more restrictive in Ice 3.5 and earlier releases:

- A data member cannot have the same name as its enclosing type:

Slice

```
class C
{
    int c; // Error
}
```

- A data member cannot have the same name as its type:

Slice

```
module M
{
    sequence<string> Seq;
    struct S
    {
        Seq Seq; // Error, use ::M::Seq as the type instead
    }
}
```

You can work around this limitation by using the fully-qualified type name.

- A parameter cannot have the same name as its operation:

Slice

```
void op(int op); // Error
```

- A parameter cannot have the same name as its type:

Slice

```
module M
{
    sequence<string> Seq;
    interface I
    {
        void op(Seq Seq); // Error, use ::M::Seq as the type
instead
    }
}
```

You can work around this limitation by using the fully-qualified type name.

See Also

- Lexical Rules

Metadata

Slice has the concept of a *metadata* directive. For example:

Slice

```
["java:type:java.util.LinkedList<Integer>"] sequence<int> IntSeq;
```

A metadata directive can appear as a prefix to any Slice definition. Metadata directives appear in a pair of square brackets and contain one or more string literals separated by commas. For example, the following is a syntactically valid metadata directive containing two strings:

Slice

```
["a", "b"] interface Example {}
```

Metadata directives are not part of the Slice language per se: the presence of a metadata directive has no effect on the client-server contract, that is, metadata directives do not change the Slice type system in any way. Instead, metadata directives are targeted at specific back-ends, such as the code generator for a particular language mapping. In the preceding example, the `java:` prefix indicates that the directive is targeted at the Java code generator.

Metadata directives permit you to provide supplementary information that does not change the Slice types being defined, but somehow influences how the compiler will generate code for these definitions. For example, a metadata directive `java:type:java.util.LinkedList<T>` instructs the Java code generator to map a sequence to a linked list instead of an array (which is the default).

Metadata directives are also used to create skeletons that support *Asynchronous Method Dispatch (AMD)*.

Apart from metadata directives that are attached to a specific definition, there are also global metadata directives. For example:

Slice

```
[["java:package:com.acme"]]
```

Note that a global metadata directive is enclosed by double square brackets, whereas a local metadata directive (one that is attached to a specific definition) is enclosed by single square brackets. Global metadata directives are used to pass instructions that affect the entire compilation unit. For example, the preceding metadata directive instructs the Java code generator to generate the contents of the source file into the Java package `com.acme`. Global metadata directives must precede any definitions in a file (but can appear following any `#include` directives).

We discuss specific metadata directives in the relevant chapters to which they apply.

You can find a summary of all metadata directives in Slice Metadata Directives.

See Also

- Slice Metadata Directives

Serializable Objects

Ice for Java and Ice for .NET allow you to send native Java and CLR objects as operation parameters. The Ice run time automatically serializes and deserializes the objects as part of an invocation. This mechanism allows you to transmit Java and CLR objects that do not have a corresponding Slice definition.

On this page:

The `serializable` Metadata Directive

To enable serialization, the parameter type must be a byte sequence with appropriate metadata. For example:

Slice

```
["java:serializable:SomePackage.JavaClass"]
sequence<byte> JavaObj;

interface JavaExample
{
    void sendJavaObj(JavaObj o);
}

["clr:serializable:SomeNamespace.CLRClass"]
sequence<byte> CLRObj;

interface CLRExample
{
    void sendCLRObj(CLRObj o);
}
```

The `java:serializable` metadata indicates that the corresponding byte sequence holds a Java serializable type named `SomePackage.JavaClass`. Your program must provide an implementation of this class; the class must implement `java.io.Serializable`.

Similarly, the `clr:serializable` metadata indicates that the corresponding byte sequences holds a CLR serializable type named `SomeNamespace.CLRClass`. Your program must provide an implementation of this class; the class must be marked with the `Serializable` attribute.

Architectural Implications

The `serializable` metadata directive permits you to transmit arbitrary Java and CLR objects across the network without the need to define corresponding Slice classes or structures. This is mainly a convenience feature: you could achieve the same thing by using ordinary Slice byte sequences and explicitly serializing your Java or CLR objects into byte sequences at the sending end, and deserializing them at the receiving end. The `serializable` metadata conveniently takes care of these chores for you and so is simpler to use.

Despite its convenience, you should use this feature with caution because it destroys language transparency. For example, a serialized Java object is useless to a C++ server. All the C++ server can do with such an object is to pass it on to some other process as a byte sequence. (Of course, if that receiving process is a Java process, it can deserialize the byte sequence.)

Further, a serialized object can be deserialized only if client and server agree on the definition of the serialized class. In Java, this is enforced by the `serialVersionUID` field of each instance; in the CLR, client and server must reference identical assembly versions. This creates much tighter coupling of client and server than exchanging Slice-defined types.

And, of course, if you build a system that relies on, for example, the exchange of serialized Java objects and you later find that you need to add C++ or C# components to the system, these components cannot do anything with the serialized Java objects other than pass them around as a blob of bytes.

So, if you do use these features, be clear that this implies tighter coupling between client and server, and that it creates additional library versioning and distribution issues because all parts of the system must agree on the implementation of the serialized objects.

See Also

- Serializable Objects in Java
- Serializable Objects in C#

Deprecating Slice Definitions

All Slice compilers support a metadata directive that allows you to deprecate a Slice definition. For example:

Slice

```
interface Example
{
    ["deprecate:someOperation() has been deprecated, use alternativeOpe
ration() instead."]
    void someOperation();

    void alternativeOperation();
}
```

The `["deprecate"]` metadata directive causes the compiler to emit code that generates a warning if you compile application code that uses a deprecated feature. This is useful if you want to remove a feature from a Slice definition but do not want to cause a hard error.

The message that follows the colon is optional; if you omit the message and use `["deprecate"]`, the Slice compilers insert a default message into the generated code.

You can apply the `["deprecate"]` metadata directive to Slice constructs other than operations (for example, a structure or sequence definition).

See Also

- Generating Slice Documentation

Using the Slice Compilers

Ice provides a separate Slice compiler for each language mapping, as shown below:

Language	Compiler
C++	`slice2cpp`
C#	`slice2cs`
Java	`slice2java`
JavaScript	`slice2js`
MATLAB	`slice2matlab`
Objective-C	`slice2objc`
PHP	`slice2php`
Python	`slice2py`
Ruby	`slice2rb`

The Slice compilers.

The compilers share a similar command-line syntax:

```
<compiler-name> [options] file...
```

Regardless of which compiler you use, a number of command-line options are common to the compilers for any language mapping. (See the appropriate language mapping chapter for options that are specific to a particular language mapping.) The common command-line options are:

- `-h, --help`
  Displays a help message.
- `-v, --version`
  Displays the compiler version.
- `-D`*`NAME`*
  Defines the preprocessor symbol *`NAME`*.
- `-D`*`NAME=DEF`*
  Defines the preprocessor symbol *`NAME`* with the value *`DEF`*.
- `-U`*`NAME`*
  Undefines the preprocessor symbol *`NAME`*.
- `-I`*`DIR`*
  Add the directory *`DIR`* to the search path for `#include` directives.
- `-E`
  Print the preprocessor output on `stdout`.
- `--output-dir` *`DIR`*
  Place the generated files into directory *`DIR`*, which must already exist.
- `-d, --debug`
  Print debug information showing the operation of the Slice parser.
- `--depend`
  Print dependency information to standard output by default, or to the file specified by the `--depend-file` option. No code is generated when this option is specified. The output includes the complete list of Slice files that the input Slice files depend on through direct or indirect inclusion; this output may include other files depending on the target programming language. The Ice for C++ build system uses the script `config/makedepend.py` to process and include this output in `Makefiles`.

- `--depend-xml`
  Print dependency information in XML format to standard output by default, or to the file specified by the `--depend-file` option. No code is generated when this option is specified. The output consists of the complete list of Slice files that the input Slice files depend on through direct or indirect inclusion, and is identical will all Slice compilers.
- `--depend-file` *`FILE`*
  Directs dependency information to the specified file. The output format depends on whether `--depend` or `--depend-xml` is also specified.
- `--validate`
  Checks the provided command-line options for correctness, and does not generate any code.

The Slice compilers permit you to compile more than a single source file, so you can compile several Slice definitions at once, for example:

```
slice2cpp -I. file1.ice file2.ice file3.ice
```

See Also

- Slice Compilation

Slice Checksums

As distributed applications evolve, developers and system administrators must be careful to ensure that deployed components are using the same client-server contract. Unfortunately, mistakes do happen, and it is not always readily apparent when they do.

To minimize the chances of this situation, the Slice compilers support an option that generates checksums for Slice definitions, thereby enabling two peers to verify that they share an identical client-server contract. The checksum for a Slice definition includes details such as parameter and member names and the order in which operations are defined, but ignores information that is not relevant to the client-server contract, such as metadata, comments, and formatting.

This option causes the Slice compiler to construct a dictionary that maps Slice type identifiers to checksums. A server typically supplies an operation that returns its checksum dictionary for the client to compare with its local version, at which point the client can take action if it discovers a mismatch.

The dictionary type is defined in the file `Ice/SliceChecksumDict.ice` as follows:

Slice

```
module Ice
{
    dictionary<string, string> SliceChecksumDict;
}
```

This type can be incorporated into an application's Slice definitions like this:

Slice

```
#include <Ice/SliceChecksumDict.ice>

module M
{
    interface MyServer
    {
        idempotent Ice::SliceChecksumDict getSliceChecksums();
        // ...
    }
}
```

The key of each element in the dictionary is a Slice type ID, and the value is the checksum of that type.

> For more information on generating and using Slice checksums, see the appropriate language mapping chapter.

See Also

- Type IDs

Generating Slice Documentation

On this page:

Documenting Slice Definitions

Adding comments to your Slice definitions is useful because it helps readers understand the semantics of your application's interfaces and data types. To make your comments more accessible, you can process your Slice files with Doxygen to produce output in HTML and other formats. Furthermore, the Slice compiler for each supported language mapping will transfer your Slice comments to the code that it generates for your Slice types, which means you can also process your generated code with a documentation generator and get meaningful results. You'll need to consider whether to generate documentation from Slice files, from generated code, or both.

The Slice API reference offers an example of the HTML output that Doxygen generates for Ice's own Slice files.

Comment Syntax

Slice uses a Javadoc-style syntax (described below) for comments. Doxygen and Slice compilers such as `slice2cpp` and `slice2java` fully support this syntax. The Slice compilers do not recognize any other Doxygen-compatible comment syntax.

As an example of the Slice comment syntax, here is the definition of `Ice::Current`:

Slice

```
/**
 *
 * Information about the current method invocation for servers. Each
 * operation on the server has a <tt>Current</tt> as its implicit final
 * parameter. <tt>Current</tt> is mostly used for Ice services. Most
 * applications ignore this parameter.
 *
 **/
local struct Current
{
    /**
     * The object adapter.
     **/
    ObjectAdapter adapter;

    /**
     * Information about the connection over which the current
     * method invocation was received. If the invocation is direct
     * due to collocation optimization, this value is set to null.
     **/
    Connection con;
```

```
    /**
     * The Ice object identity.
     **/
    Identity id;

    /**
     * The facet.
     ***/
    string facet;

    /**
     * The operation name.
     **/
    string operation;

    /**
     * The mode of the operation.
     **/
    OperationMode mode;

    /**
     * The request context, as received from the client.
     **/
    Context ctx;

    /**
     * The request id unless oneway (0) or collocated (-1).
     **/
    int requestId;

    /**
     * The encoding version used to encode the input and output
parameters.
     **/
```

```
    EncodingVersion encoding;
}
```

If you look at the comments, you will see these reflected in the documentation for `Ice::Current` in the online Slice API reference.

A documentation comment starts with `/**` and ends with `**/`. Such a comment can precede any Slice construct, such as a module, interface, structure, operation, and so on. Within a documentation comment, you can either start each line with a `*`, or you can leave the beginning of the line blank:

Slice

```
/**
 *
 * This is a documentation comment for which every line
 * starts with a '*' character.
 **/

/**

 This is a documentation comment without a leading '*'
 for each line. Either style of comment is fine.

 **/
```

The first sentence of the documentation comment for a Slice construct should be a summary sentence.

Hyperlinks

Any Slice identifier enclosed in `{@link ...}` is presented as a hyperlink in code font. For example:

Slice

```
/**
 * An empty {@link name} denotes a null object.
 **/
```

This generates a hyperlink for the `name` markup that points at the definition of the corresponding Slice symbol. (The symbol can denote any Slice construct, such as a type, interface, parameter, or structure member.)

> Doxygen by default will automatically create links from symbols that appear in comments, which means explicit `@link` tags are rarely necessary in practice.

Explicit Cross References

The directive `@see` creates an explicit cross reference to another entity:

Slice

```
/**
 * The object adapter, which is responsible for receiving requests
 * from endpoints, and for mapping between servants, identities,
 * and proxies.
 *
 * @see Communicator
 * @see ServantLocator
 **/
```

Markup for Operations

There are three directives specifically to document Slice operations: `@param`, `@return`, and `@throws`. For example:

Slice

```
/**
 * Look for an item with the specified
 * primary and secondary key.
 *
 * @param p The primary search key.
 *
 * @param s The secondary search key.
 *
 * @return The item that matches the specified keys.
 *
 * @throws NotFound Raised if no item matches the specified keys.
 **/

Item findItem(Key p, Key s) throws NotFound;
```

For clarity, the comment order should match the order of declaration for the parameters.

General HTML Markup

A documentation comment can contain any markup that is permitted by HTML in that place, such as `<table>` or `<ul>` elements. Please see the HTML specification for details.

Using Doxygen for Slice Documentation

This section describes how to use Doxygen to generate a Slice API reference documentation from your Slice files.

Selecting a Doxygen Version

The current version of Doxygen does not recognize Slice (`*.ice`) files by default. Furthermore, the way its output presents Slice types is not ideal. For example, modules are called namespaces; structs, interfaces and exceptions are grouped together as classes; sequences and dictionaries are grouped together as typedefs; and so on. If you still want to use this version of Doxygen, you can force it to recognize Slice

files by using the `EXTENSION_MAPPING` and `FILE_PATTERNS` settings:

```
EXTENSION_MAPPING = ice=c
FILE_PATTERNS = *.ice
```

Instead, we recommend using ZeroC's fork of Doxygen that automatically recognizes Slice files and provides a Slice-optimized output mode. The Slice API reference was generated using this mode.

Configuring Doxygen

You can generate a default configuration file as follows:

```
doxygen -g config
```

Next you'll want to edit the generated file (in the example above we called the file `config`) and review its settings.

At a minimum, we recommend that you review the following settings, shown in the order they appear in the file:

- `PROJECT_NAME`
  The project name is used as the page title.
- `JAVADOC_AUTOBRIEF`
  This setting is disabled by default. Enable it to use the first sentence of a construct's documentation as its summary.
- `OPTIMIZE_OUTPUT_SLICE`
  This setting is only available with ZeroC's fork of Doxygen. Enabling it changes the output in several ways: modules are called "Modules" instead of "Namespaces"; classes, interfaces, structs and exceptions have separate indices; sequences and dictionaries have their own sections; and constants are called "Constants" instead of "Variables". The Slice API reference demonstrates these changes.
- `EXTRACT_ALL`
  Enabling this setting causes Doxygen to include all constructs even if they aren't documented.
- `SORT_BRIEF_DOCS`
  This setting is disabled by default but we recommend enabling it, which causes the summary sections of a module to be sorted alphabetically rather than in the order of declaration.
- `INPUT`
  List the files or subdirectories to be processed.
- `RECURSIVE`
  Enable this setting to force Doxygen to recurse into subdirectories when searching for input files.
- `EXCLUDE_SYMBOLS`
  Use this setting to exclude certain types or modules that you don't want to appear in the output.
- `USE_MDFILE_AS_MAINPAGE`
  This setting allows you to specify the name of a markdown file that contains content for the main page of the output. Note that this file must also be mentioned in `INPUT`.
- `COLS_IN_ALPHA_INDEX`
  The default value of 5 for this setting can make for an overly wide presentation. We recommend using 3 instead.
- `GENERATE_LATEX`
  LaTeX output is enabled by default. If you don't need to generate this format, set this to `NO`.
- `INCLUDE_PATH`
  Specify the path name of any include directories used by your Slice files.
- `TAGFILES`
  Refer to the next section below.
- `HAVE_DOT`
  Enable this setting if you want Doxygen to use the `dot` tool to generate inheritance and collaboration diagrams.

When you're ready to generate documentation, run Doxygen like this:

```
doxygen config
```

By default, the HTML output will be generated into an `html` subdirectory.

Linking to ZeroC Documentation

Doxygen has the ability to embed links to the documentation of external types that are used by your source files. For example, if your Slice file refers to a type from the Ice API reference and you'd like your documentation to link to ZeroC's documentation for that type, you simply need to download the appropriate *tag file* for your Ice version and configure the `TAGFILES` setting in your Doxygen configuration file:

```
TAGFILES = slice.tag=https://doc.zeroc.com/api/Ice37/slice
```

The tag file is available here:

- https://doc.zeroc.com/api/Ice37/slice/slice.tag

See Also

- Slice API reference
- HTML specification

Slice Keywords

The following identifiers are Slice keywords:

`bool`	`extends`	`LocalObject`	`string`
`byte`	`false`	`long`	`struct`
`class`	`float`	`module`	`throws`
`const`	`idempotent`	`Object`	`true`
`dictionary`	`implements`	`optional`	`Value`
`double`	`int`	`out`	`void`
`enum`	`interface`	`sequence`	
`exception`	`local`	`short`	

Keywords must be capitalized as shown.

See Also

- Lexical Rules

Slice Metadata Directives

On this page:

General Metadata Directives

`amd`

This directive applies to interfaces, classes, and individual operations. It enables code generation for asynchronous method dispatch. (See the relevant language mapping chapter for details.)

`delegate`

This directive applies only to local interfaces with one operation. Interfaces with this metadata will be generated as a `std::function` in C++, a `delegate` in C#, and an interface with one operation (usable as a `FunctionalInterface` for Java 8) in Java.

`deprecate`

This directive allows you to emit a deprecation warning for Slice constructs.

`ice-prefix`

This global directive allows the use of identifiers that start with the reserved prefix `Ice` (or `ICE`, `ice` etc.). Only Slice files provided by Ice should use this directive.

`format`

This directive defines the encoding format used for any classes or exceptions marshaled as the arguments or results of an operation. The tag can be applied to an interface, which affects all of its operations, or to individual operations. Legal values for the tag are `format:sliced`, `format:compact`, and `format:default`. A tag specified for an operation overrides any setting applied to its enclosing interface. The `Ice.Default.SlicedFormat` property defines the behavior when no tag is present.

`marshaled-result`

This directive changes the return type of servant methods so that a servant can force its results to be marshaled immediately in a thread-safe manner. Refer to the relevant server-side language mapping sections for more information on the rules for parameter passing.

`preserve-slice`

This directive applies to classes and exceptions, allowing an intermediary to forward an instance of the annotated type, or any of its subtypes, with all of its slices intact. Operations that transfer such types must be annotated with `format:sliced`. It is not necessary to repeat the `preserve-slice` tag on derived types, but you may wish to do so for documentation purposes.

`protected`

This directive applies to data members of classes and changes code generation to make these members protected. See class mapping of the relevant language mapping chapter for more information.

`suppress-warning`

This global directive allows to suppress Slice compiler warnings. It applies to all definitions in the Slice file that includes this directive. If one or more categories are specified (for example "`suppress-warning:invalid-metadata`" or "`suppress-warning:deprecated, invalid-metadata`") only warnings matching these categories will be suppressed, otherwise all warnings are suppressed. The categories are described in the following table:

Suppress Warning Category	Description
`all`	Suppress all Slice compiler warnings. Equivalent to `[["suppress-warning"]]`.

`deprecated`	Suppress warnings related to deprecated features.
`invalid-metadata`	Suppress warnings related to invalid metadata.

`underscore`

This global directive allows the use of identifiers with underscores. It applies to all definitions in the Slice file that includes this directive.

Metadata Directives for C++

The Slice to C++ compiler understands three C++ metadata prefixes: `cpp`, `cpp11` and `cpp98`. Most C++ metadata directives can be specified with either one of the prefixes.

`cpp11` and `cpp98` directives applies only to their respective mapping. A `cpp` metadata directive applies to both mappings, and can be overridden by the same directive with `cpp98` or `cpp11`. For example:

Slice

```
["cpp:type:MyType", "cpp11:type:MyNewType"] sequence<byte> ByteSeq;
```

maps `ByteSeq` to `MyType` with the C++98 mapping, and to `MyNewType` with the C++11 mapping.

`cpp:array`

This directive applies to sequence parameters in operations. It directs the code generator to map these parameters to pairs of pointers.

`cpp:class`

This directive applies to structures. It directs the code generator to create a C++ class (instead of a C++ structure) to represent a Slice structure. This is a C++98-only metadata: it has no effect with the C++11 mapping.

`cpp:comparable`

This directive applies to structures. It directs the code generator to generate comparison operators for a structure regardless of whether it qualifies as a legal dictionary key type. This is a C++98-only metadata: it has no effect with the C++11 mapping.

`cpp:const`

This directive applies to operations. It directs the code generator to create a `const` pure virtual member function for the skeleton class.

`cpp:dll-export:SYMBOL`

This global directive applies to all definitions in a Slice file.

Use *`SYMBOL`* to control the export and import of symbols from DLLs on Windows and dynamic shared libraries on other platforms. This option allows you to export symbols from the generated code, and place such generated code in a DLL (on Windows) or shared library (on other platforms). As an example, compiling a Slice file `Widget.ice` with:

Slice

```
[["cpp:dll-export:WIDGET_API"]]
```

results in the following additional code being generated into `Widget.h`:

C++

```
#ifndef WIDGET_API
#   if defined(ICE_STATIC_LIBS)
#       define WIDGET_API /**/
#   ifdef WIDGET_API_EXPORTS
#       define WIDGET_API ICE_DECLSPEC_EXPORT
#   else
#       define WIDGET_API ICE_DECLSPEC_IMPORT
#   endif
#endif
```

The generated code also includes the provided `SYMBOL` name (`WIDGET_API` in our example) in the declaration of classes and functions that need to be exported (when building a DLL or dynamic library) or imported (when using such library).

`ICE_DECLSPEC_EXPORT` and `ICE_DECLSPEC_IMPORT` are macros that expand to compiler-specific attributes. For example, for Visual Studio, they are defined as:

C++

```
#if defined(_MSC_VER)
#   define ICE_DECLSPEC_EXPORT __declspec(dllexport)
#   define ICE_DECLSPEC_IMPORT __declspec(dllimport)
```

With GCC and clang, they are defined as:

C++

```
#elif defined(__GNUC__) || defined(__clang__)
#   define ICE_DECLSPEC_EXPORT __attribute__((visibility ("default")))
#   define ICE_DECLSPEC_IMPORT __attribute__((visibility ("default")))
```

The generated .cpp file (`Widget.cpp` in our example) defines `SYMBOL_EXPORTS`; this way, you don't need to do anything special when compiling generated files.

cpp:header-ext

This global directive allows you to use a file extension for C++ header files other than the default `.h` extension.

cpp:ice_print

This directive applies to exceptions. It directs the code generator to declare (but not implement) an `ice_print` member function that overrides the `ice_print` virtual function inherited from an Ice base class. The application must provide the implementation of this `ice_print` function.

cpp:include

This global directive allows you inject additional `#include` directives into the generated code. This is useful for custom types.

cpp:noexcept

This directive applies only to operations on local interfaces. When specified, the generated C++ pure virtual function declaration will carry the `noexcept` specifier (C++11) or the `ICE_NOEXCEPT macro` (C++98). ICE_NOEXCEPT expands to `noexcept` or `throw()` depending the C++ compiler and C++ compilation flags.

cpp:range

This directive applies to sequence parameters in operations. It directs the code generator to map these parameters to pairs of iterators. This is a C++98-only metadata direcetive: it has no effect with the C++11 mapping.

cpp:scoped

This directive applies to enumerations. It directs the code generator to use the enumeration's name as prefix for all generated C++ enumerators. This is a C++98-only metadata directive: it has no effect on the C++11 mapping. See also `cpp:unscoped` below.

cpp:source-ext

This global directive allows you to use a file extension for C++ source files other than the default `.cpp` extension.

cpp:type:*c++-type*

This directive applies to sequences and dictionaries. It directs the code generator to map the Slice type or parameter to the provided C++ type.

cpp:type:string and cpp:type:wstring

These directives apply to data members of type string as well as to containers, such as structures, classes and exceptions. String members map by default to `std::string`. You can use the `cpp:type:wstring` metadata to cause a string data member (or all string data members in a structure, class or exception) to map to `std::wstring` instead. Use the `cpp:type:string` metadata to force string members to use the default mapping regardless of any enclosing metadata.

Slice

```
module A
{
    ["cpp:type:wstring"] struct Struct1
    {
        string s1; // Maps to std::wstring
        ["cpp:type:string"] string s2; // Maps to std::string
    }
}
```

cpp:unscoped

This directive applies to enumerations. It directs the code generator to create an old-type unscoped C++ enumeration instead of a scoped enumeration (`enum class`). This is a C++11-only directive: it has no effect on the C++98 mapping. See also `cpp:scoped` above.

cpp:view-type:*c++-view-type*

This directive applies to string, sequence and dictionary parameters. It directs the code generator to map this parameter to the provided C++ type when this parameter does not need to hold any memory, for example when mapping an in-parameter to a proxy function.

`cpp:virtual`

This directive applies to classes with the C++98 mapping only. It has no effect with the C++11 mapping. If the directive is present and a class has base classes, the generated C++ class derives virtually from its bases; without this directive, slice2cpp generates the class so it derives non-virtually from its bases.

This directive is useful if you use Slice classes as servants and want to inherit the implementation of operations in the base class in the derived class. For example:

Slice

```
class Base
{
    int baseOp();
}

["cpp:virtual"]
class Derived extends Base
{
    string derivedOp();
}
```

The metadata directive causes slice2cpp to generate the class definition for `Derived` using virtual inheritance:

C++98

```
class Base : public virtual Ice::Object
{
    // ...
};

class Derived : public virtual Base
{
    // ...
};
```

This allows you to reuse the implementation of `baseOp` in the servant for `Derived` using ladder inheritance:

C++98

```
class BaseI : public virtual Base
{
    Ice::Int baseOp(const Ice::Current&);
    // ...
};

class DerivedI : public virtual Derived, public virtual BaseI
{
    // Re-use inherited baseOp()
};
```

Note that, if you have data member in classes and use virtual inheritance, you need to take care to correctly call base class constructors if you implement your own one-shot constructor. For example:

Slice

```
class Base
{
    int baseInt;
}

class Derived extends Base
{
    int derivedInt;
}
```

The generated one-shot constructor for `Derived` initializes both `baseInt` and `derivedInt`:

C++98

```
Derived::Derived(Ice::Int iceP_baseInt, Ice::Int iceP_derivedInt) :
    M::Base(iceP_baseInt),
    derivedInt(iceP_derivedInt)
{
}
```

If you derive your own class from `Derived` and add a one-shot constructor to your class, you must explicitly call the constructor of all the base classes, including `Base`. Failure to call the `Base` constructor will result in `Base` being default-constructed instead of getting a defined value. For example:

C++98

```
class DerivedI : public virtual Derived
{
public:
    DerivedI(int baseInt, int derivedInt, const string& s) :
        Base(baseInt), Derived(baseInt, derivedInt), _s(s)
    {
    }

private:
    string _s;
};
```

This code correctly initializes the `baseInt` member of the `Base` part of the class. Note that the following does not work as intended and leaves the `Base` part default-constructed (meaning that `baseInt` is not initialized):

C++98

```
class DerivedI : public virtual Derived
{
public:
    DerivedI(int baseInt, int derivedInt, const string& s) :
        Derived(baseInt, derivedInt), _s(s)
    {
        // WRONG: Base::baseInt is not initialized.
    }

private:
    string _s;
};
```

Metadata Directives for C#

The metadata for C# (or .NET) directives uses the `cs` prefix or the `clr` prefix. These two prefixes are interchangeable: `cs:attribute` and `clr:attribute` have exactly the same meaning. We present these directives with the `cs` prefix below.

In Ice releases prior to 3.7, the `cs` prefix was used exclusively for the metadata directives `cs:attribute` and `cs:tie` while the `clr` prefix was used for all other directives.

`cs:attribute`

This directive can be used both as a global directive and as directive for specific Slice constructs. It injects C# attribute definitions into the generated code. (See C-Sharp Attribute Metadata Directive.)

`cs:class`

This directive applies to Slice structures. It directs the code generator to emit a C# class instead of a structure.

`cs:generic:List`, `cs:generic:LinkedList`, `cs:generic:Queue` and `cs:generic:Stack`

These directives apply to sequences and map them to the specified sequence type.

`cs:generic:SortedDictionary`

This directive applies to dictionaries and maps them to `SortedDictionary`.

`cs:generic`

This directive applies to sequences and allows you map them to custom types.

`cs:implements:`*`type`*

This directive adds the specified base type to the generated code for a Slice structure, class or interface. For example, Ice defines the `Communicator` interface as shown below:

Slice

```
["cs:implements:_System.IDisposable"]
local interface Communicator { ... }
```

Consequently, the generated C# interface `Ice.Communicator` implements `IDisposable`.

Every Slice-generated C# source file defines two namespace aliases:

```
using _System = global::System;
using _Microsoft = global::Microsoft;
```

We recommend using these aliases if your metadata refers to the `System` or `Microsoft` namespaces.

When used with structs, this metadata can only refer to interfaces without operations. With classes, the code is responsible for registering a value factory if the Slice class is transferred over-the-wire and uses this metadata to implement native C# interfaces.

cs:property

This directive applies to Slice structures and classes. It directs the code generator to create C# property definitions for data members.

cs:serializable

This directive allows you to use Ice to transmit serializable CLR classes as native objects, without having to define corresponding Slice definitions for these classes.

cs:tie

This directive applies to an interface or a class with operations, and triggers the generation of a tie class.

Metadata Directives for Java

java:buffer

This directive applies to sequences of certain primitive types. It directs the translator to map the sequence to a subclass of `java.nio.Buffer`.

java:getset

This directive applies to data members and structures, classes, and exceptions. It adds accessor and modifier methods (JavaBean methods) for data members.

java:implements:*type*

This directive adds the specified base interface to the generated code for a Slice structure, class or interface. For example, Ice defines the `Communicator` interface as shown below:

Slice

```
["java:implements:java.lang.AutoCloseable"]
local interface Communicator { ... }
```

Consequently, the generated Java interface `Communicator` implements `java.lang.AutoCloseable`.

When used with structs, this metadata can only refer to interfaces without operations. With classes or interfaces, the generated Java interface will be marked as `abstract`. The code is responsible for registering a value factory if the Slice class is transferred over-the-wire and uses this metadata to implement native Java interfaces.

`java:optional`

> This directive is only used by the Java Compat mapping and has no effect in the Java mapping.

This directive forces optional output parameters to use the optional mapping instead of the default required mapping in servants.

`java:package`

This directive instructs the code generator to place the generated classes into a specific package. The directive can be used as global metadata and can also be applied to top-level modules. Global metadata (if any) serves as the default directive unless overridden by module metadata.

When applying this directive to a module that is reopened by your Slice file, you must repeat the metadata for each reopening of the module if you want the packaging to remain consistent:

Slice

```
["java:package:MyPackage"]
module M
{
    ...
}

//
// Re-open module M to define more types
//
["java:package:MyPackage"]
module M
{
    ...
}
```

`java:serializable`

This directive allows you to use Ice to transmit serializable Java classes as native objects, without having to define corresponding Slice

definitions for these classes.

java:serialVersionUID

This directive overrides the default (generated) value of `serialVersionUID` for a Slice type.

java:tie

> This directive is only used by the Java Compat mapping and has no effect in the Java mapping.

This directive applies to an interface or a class with operations, and triggers the generation of a tie class.

java:type

This directive allows you to use custom types for sequences and dictionaries.

java:UserException

This directive applies to operations, and indicates that the generated Java methods on the mapped servant interfaces and class can throw any user exception, regardless of its specific definition. The exception specification for these methods is simply `throws com.zeroc.Ice.UserException` (Java) or `throws Ice.UserException` (Java Compat). This metadata has no effect on the methods of generated proxies. The directive `UserException` (without the `java:` prefix) is a deprecated alias for `java:UserException`.

Metadata Directives for Objective-C

objc:dll-export:SYMBOL

This global directive applies to all definitions in a Slice file.

Use *`SYMBOL`* to control the export of symbols from dynamic shared libraries. This option allows you to export symbols from the generated code and place such generated code in a shared library. Compiling a Slice definition with:

Slice

```
[["objc:dll-export:WIDGET_API"]]
```

adds the provided `SYMBOL` name (`WIDGET_API` in our example) to the declaration of interfaces and protocols that need to be exported. The generated code also defines the provided SYMBOL as `__attribute__((visibility ("default")))`.

> This option is useful when you create a shared library and compile your Objective-C code with `-fvisibility=hidden` to reduce the number of symbols exported.

objc:header-dir

This global directive allows you to specify a prefix for the header path in generated Objective-C files. For example, all the Slice files from Ice set this metadata to `objc` to allow importing Objective-C headers from the `objc` directory (e.g.: `#import <objc/Ice/Ice.h>`).

objc:prefix

This directive applies to modules and changes the default mapping for modules to use a specified prefix.

`objc:scoped`

This directive applies to enumerations. It directs the code generator to use the enumeration's name as prefix for all generated Objective-C enumerators.

Metadata Directives for Python

`python:package`

This directive instructs the code generator to enclose the generated code in a specified Python package. The directive can be used as global metadata and can also be applied to top-level modules. Global metadata (if any) serves as the default directive unless overridden by module metadata.

When applying this directive to a module that is reopened by your Slice file, you must repeat the metadata for each reopening of the module if you want the packaging to remain consistent:

Slice

```
["python:package:MyPackage"]
module M
{
    ...
}

//
// Re-open module M to define more types
//
["python:package:MyPackage"]
module M
{
    ...
}
```

`python:pkgdir`

This global directive instructs the code generator to place the generated code into a specified directory.

`python:seq:default`, `python:seq:list` and `python:seq:tuple`

These directives allow you to change the mapping for Slice sequences.

Metadata Directives for Freeze

`freeze:read` and `freeze:write`

These directives inform a Freeze evictor whether an operation updates the state of an object, so the evictor knows whether it must save an object before evicting it.

See Also

- Metadata

Slice for a Simple File System

For this manual, we use a file system application to illustrate various aspects of Ice. Throughout, we progressively improve and modify the application such that it evolves into an application that is realistic and illustrates the architectural and coding aspects of Ice. This allows us to explore the capabilities of the platform to a realistic degree of complexity without overwhelming you with an inordinate amount of detail early on.

In this section:

- File System Application outlines the file system functionality
- Slice Definitions for the File System develops the data types and interfaces that are required for the file system
- Complete Definition presents the complete Slice definition for the application.

File System Application

Our file system application implements a simple hierarchical file system, similar to the file systems we find in Windows or Unix. To keep code examples to manageable size, we ignore many aspects of a real file system, such as ownership, permissions, symbolic links, and a number of other features. However, we build enough functionality to illustrate how you could implement a fully-featured file system, and we pay attention to things such as performance and scalability. In this way, we can create an application that presents us with real-world complexity without getting buried in large amounts of code.

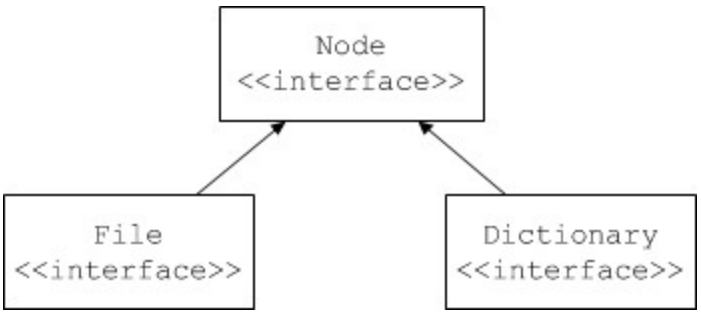
Our file system consists of directories and files. Directories are containers that can contain either directories or files, meaning that the file system is hierarchical. A dedicated directory is at the root of the file system. Each directory and file has a name. Files and directories with a common parent directory must have different names (but files and directories with different parent directories can have the same name). In other words, directories form a naming scope, and entries with a single directory must have unique names. Directories allow you to list their contents.

For now, we do not have a concept of pathnames, or the creation and destruction of files and directories. Instead, the server provides a fixed number of directories and files. (We will address the creation and destruction of files and directories in Object Life Cycle.)

Files can be read and written but, for now, reading and writing always replace the entire contents of a file; it is impossible to read or write only parts of a file.

Slice Definitions for the File System

Given the very simple requirements we just outlined, we can start designing interfaces for the system. Files and directories have something in common: they have a name and both files and directories can be contained in directories. This suggests a design that uses a base type that provides the common functionality, and derived types that provide the functionality specific to directories and files, as shown:



Inheritance Diagram of the File System.

The Slice definitions for this look as follows:

Slice

```
module Filesystem
{
    interface Node
    {
        // ...
    }

    interface File extends Node
    {
        // ...
    }

    interface Directory extends Node
    {
        // ...
    }
}
```

Next, we need to think about what operations should be provided by each interface. Seeing that directories and files have names, we can add an operation to obtain the name of a directory or file to the `Node` base interface:

Slice

```
interface Node
{
    idempotent string name();
}
```

The `File` interface provides operations to read and write a file. For simplicity, we limit ourselves to text files and we assume that `read` operations never fail and that only `write` operations can encounter error conditions. This leads to the following definitions:

Slice

```
exception GenericError
{
    string reason;
}

sequence<string> Lines;

interface File extends Node
{
    idempotent Lines read();
    idempotent void write (Lines text) throws GenericError;
}
```

Note that `read` and `write` are marked idempotent because either operation can safely be invoked with the same parameter value twice in a row: the net result of doing so is the same has having (successfully) called the operation only once.

The `write` operation can raise an exception of type `GenericError`. The exception contains a single `reason` data member, of type `string`. If a `write` operation fails for some reason (such as running out of file system space), the operation throws a `GenericError` exception, with an explanation of the cause of the failure provided in the `reason` data member.

Directories provide an operation to list their contents. Because directories can contain both directories and files, we take advantage of the polymorphism provided by the `Node` base interface:

Slice

```
sequence<Node*> NodeSeq;

interface Directory extends Node
{
    idempotent NodeSeq list();
}
```

The `NodeSeq` sequence contains elements of type `Node*`. Because `Node` is a base interface of both `Directory` and `File`, the `NodeSeq` sequence can contain proxies of either type. (Obviously, the receiver of a `NodeSeq` must down-cast each element to either `File` or `Directory` in order to get at the operations provided by the derived interfaces; only the `name` operation in the `Node` base interface can be invoked directly, without doing a down-cast first. Note that, because the elements of `NodeSeq` are of type `Node*` (not `Node`), we are using pass-by-reference semantics: the values returned by the `list` operation are proxies that each point to a remote node on the server.

These definitions are sufficient to build a simple (but functional) file system. Obviously, there are still some unanswered questions, such as how a client obtains the proxy for the root directory. We will address these questions in the relevant implementation chapter.

Complete Definition

We wrap our definitions in a module, resulting in the final definition as follows:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

See Also

- Object Life Cycle

Language Mappings

Topics

C++11 Mapping

Topics

Selecting the C++11 Mapping

Ice provides two distinct C++ mappings:

- C++98
  This was the only C++ mapping provided by Ice until version 3.7. This mapping relies only on features present in the ISO/IEC 14882:1998 C++ standard, informally known as C++98. It includes its own helper classes for smart pointers, threads, mutexes and so on.

- C++11
  This is a new mapping that takes advantage of features in the ISO/IEC 14882:2011 C++ standard, and occasionally newer features. This mapping requires a recent C++ compiler in C++11 or C++14/17 mode.

This chapter describes the C++11 mapping.

slice2cpp, the Slice-to-C++ translator, always generates code for both mappings, and C++ headers files provided by Ice, such as `Ice/Ice.h` and `IceGrid/IceGrid.h`, can be used with either mapping.

Selecting the C++11 Mapping

You select the C++11 mapping by compiling all your code with `-DICE_CPP11_MAPPING`. This macro should be defined in your build projects, not in your source files.

You also need to link your application with the `++11` variant of the Ice libraries, for example:

```
$ c++ -o client Hello.cpp client.cpp -DICE_CPP11_MAPPING -lIce++11.so
```

> The Ice C++11 and Ice C++98 libraries are built from the same source code, in ice/cpp. The resulting C++ libraries are nevertheless language mapping specific: `libIce++11.so` is for the Ice C++11 mapping, while `libIce.so` is for the Ice C++98 mapping.

Initialization and CommunicatorHolder in C++11

On this page:

Initializing the Ice Run Time with `Ice::initialize`

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `Ice::Communicator` object.

A `Communicator` is a local C++ object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application or program.

You initialize the Ice run time by calling the C++ function `Ice::initialize`, for example:

C++

```
int
main(int argc, char* argv[])
{
    std::shared_ptr<Ice::Communicator> communicator
= Ice::initialize(argc, argv);
    // ...
}
```

`initialize` accepts a C++ reference to `argc` and an argument vector `argv`. The function scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

> `Ice::initialize` has additional overloads to permit other information to be passed to the Ice run time.

`initialize` is a low-level function, as you need to explicitly call `destroy` on the returned `Communicator` object when you're done with Ice, typically just before returning from `main`. The `destroy` member function is responsible for finalizing the Ice run time. In particular, in a server, `destroy` waits for any operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory.

The general shape of the `main` function of an Ice-based application is therefore:

C++

```
#include <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    int status = 0;
    try
    {
        // communicator is std::shared_ptr<Ice::Communicator>
        auto communicator = Ice::initialize(argc, argv);

        try
        {
            ... application code ...

            communicator->destroy(); // destroy is noexcept
        }
        catch(const std::exception&)
        {
            ...
            // make sure communicator is destroyed if an exception is
thrown
            communicator->destroy();
            throw;
        }
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        status = 1;
    }
    return status;
}
```

This code is a little bit clunky, as we need to make sure the communicator gets destroyed in all paths, including when an exception is thrown. As a result, most of the time, you should not call `initialize` directly: you should use instead a helper class that calls `initialize` and ensures the resulting communicator gets eventually destroyed.

`Ice::CommunicatorHolder` RAII Helper Class

A `CommunicatorHolder` is a small RAII helper class that creates a `Communicator` in its constructor (by calling `initialize`) and destroys this communicator in its destructor.

With a `CommunicatorHolder`, our typical `main` function becomes much simpler:

C++

```
#include <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    int status = 0;
    try
    {
        Ice::CommunicatorHolder ich(argc, argv); // Calls
Ice::initialize

        ... application code ...

        // CommunicatorHolder's destructor calls destroy on the
communicator
        // whether or not an exception is thrown
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        status = 1;
    }
    return status;
}
```

`Ice::CommunicatorHolder` is defined as follows:

C++

```
namespace Ice
{
    class CommunicatorHolder
    {
    public:

        CommunicatorHolder();
        template<class... T> explicit CommunicatorHolder(T&&... params);

        CommunicatorHolder(std::shared_ptr<Communicator>);
        CommunictorHolder& operator=(std::shared_ptr<Communicator>);

        CommunicatorHolder(const CommunicatorHolder&) = delete;
        CommunicatorHolder(CommunicatorHolder&&) = default;
        CommunicatorHolder& operator=(CommunicatorHolder&&);

        ~CommunicatorHolder();

        explicit operator bool() const;
        const std::shared_ptr<Communicator>& communicator() const;
        const std::shared_ptr<Communicator>& operator->() const;
        std::shared_ptr<Communicator> release();
        ...
    };
}
```

Let's examine each of these functions:

- `CommunicatorHolder()`
  This default constructor creates an empty holder (holds no `Communicator`).
- `template<class... T> explicit CommunicatorHolder(T&&... params)`
  These constructors call `initialize` with the provided parameters; the new `CommunicatorHolder` then holds the resulting `Communicator` in a private data member (not shown).
- `CommunicatorHolder(std::shared_ptr<Communicator>)`
  This constructor adopts the given communicator: the `CommunicatorHolder` becomes responsible for calling `destroy` on it.
- `CommunicatorHolder& operator=(std::shared_ptr<Communicator>)`
  This assignment operator destroys the communicator held by this `CommunicatorHolder`, then adopts the provided communicator.
- `CommunicatorHolder& operator=(CommunicatorHolder&&)`
  This assignment operator destroys the communicator held by this `CommunicatorHolder`, then adopts the other `CommunicatorHolder`'s communicator.
- `~CommunicatorHolder()`
  The destructor calls `destroy` on the communicator held by this `CommunicatorHolder`.
- `explicit operator bool() const`
  Returns `true` when this `CommunicatorHolder` holds a `Communicator`, and `false` otherwise.
- `const std::shared_ptr<Communicator>& communicator() const`

  This function gives read-only access to the communicator held by this `CommunicatorHolder`.

- `const std::shared_ptr<Communicator>& operator->() const`
  This arrow operator allows you to use a `CommunicatorHolder` just like a `Communicator` object. For example:

```
C++
Ice::CommunicatorHolder ich(argc, argv);
auto base = ich->stringToProxy("SimplePrinter:default -p 10000");
```

  is equivalent to:

```
C++
Ice::CommunicatorHolder ich(argc, argv);
auto base =
ich->communicator()->stringToProxy("SimplePrinter:default -p
10000");
```

- `std::shared_ptr<Communicator> release()`
  This function returns the communicator held by `CommunicatorHolder` to the caller, and the caller becomes responsible for destroying this communicator. `CommunicatorHolder` no longer holds a communicator after this call.

The default constructor of `CommunicatorHolder` does nothing:

```
C++
Ice::CommunicatorHolder ich; // does not create a Communicator,
ich.communicator() returns a null shared_ptr
```

If you want to create a `CommunicatorHolder` that holds a `Communicator` created by `initialize` with no args, you can write:

```
C++
Ice::CommunicatorHolder ich = Ice::initialize();
```

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-C++11 Mapping

The client-side Slice-to-C++ mapping defines how Slice data types are translated to C++ types, and how clients invoke operations, pass parameters, and handle errors. Much of the C++ mapping is intuitive. For example, Slice sequences map to STL vectors, so there is essentially nothing new you have to learn in order to use Slice sequences in C++.

The rules that make up the C++ mapping are simple and regular. In particular, the mapping is free from the potential pitfalls of memory management: all types are self-managed and automatically clean up when instances go out of scope. This means that you cannot accidentally introduce a memory leak by, for example, ignoring the return value of an operation invocation or forgetting to deallocate memory that was allocated by a called operation.

The C++ mapping is fully thread-safe. Nevertheless, you still need to synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. You only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the C++ mapping, you should need no more than the Slice definition of your application and knowledge of the C++ mapping rules. In particular, looking through the generated header files in order to discern how to use the C++ mapping is likely to be confusing because the header files are not necessarily meant for human consumption and, occasionally, contain various cryptic constructs to deal with operating system and compiler idiosyncrasies. Of course, occasionally, you may want to refer to a header file to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **The `Ice` Namespace**
> All of the APIs for the Ice run time are nested in the `Ice` namespace, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` namespace are generated from Slice definitions; other parts of the `Ice` namespace provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` namespace throughout the remainder of the manual.

Topics

C++11 Mapping for Identifiers

A Slice identifier maps to an identical C++ identifier. For example, the Slice identifier `Clock` becomes the C++ identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a C++ keyword, the corresponding C++ identifier is prefixed with `_cpp_`. For example, the Slice identifier `while` is mapped as `_cpp_while`.

A single Slice identifier often results in several C++ identifiers. For example, for a Slice interface named `Foo`, the generated C++ code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_cpp_while` and `while Prx` (*not* `_cpp_whilePrx`), that is, the prefix is applied only to those generated identifiers that actually require it.

> You should try to avoid such identifiers as much as possible.

See Also

- Lexical Rules
- C++11 Mapping for Modules
- C++11 Mapping for Built-In Types
- C++11 Mapping for Enumerations
- C++11 Mapping for Structures
- C++11 Mapping for Sequences
- C++11 Mapping for Dictionaries
- C++11 Mapping for Constants
- C++11 Mapping for Exceptions

C++11 Mapping for Modules

A Slice module maps to a C++ namespace. The mapping preserves the nesting of the Slice definitions. For example:

Slice

```
module M1
{
    module M2
    {
        // ...
    }
    // ...
}

// ...

module M1    // Reopen M1
{
    // ...
}
```

This definition maps to the corresponding C++ definition:

C++

```
namespace M1
{
    namespace M2
    {
        // ...
    }
    // ...
}

// ...

namespace M1    // Reopen M1
{
    // ...
}
```

If a Slice module is reopened, the corresponding C++ namespace is reopened as well.

See Also

- Modules
- C++11 Mapping for Identifiers
- C++11 Mapping for Built-In Types
- C++11 Mapping for Enumerations
- C++11 Mapping for Structures
- C++11 Mapping for Sequences

C++11 Mapping for Built-In Types

On this page:

Mapping of Slice Built-In Types to C++ Types

The Slice built-in types are mapped to C++ types as shown in this table:

Slice	C++
bool	`bool`
byte	`Ice::Byte`
short	`short`
int	`int`
long	`long long`
float	`float`
double	`double`
string	`std::string`

> `Ice::Byte` is a typedef for `unsigned char`. This guarantees that byte values are always in the range 0..255.

Alternative String Mapping for C++

You can use a metadata directive, `"cpp:type:wstring"`, to map strings to C++ `std::wstring`. For containers (such as interfaces or structures), the metadata directive applies to all strings within the container. A corresponding metadata directive, `"cpp:type:string"`, can be used to selectively override the mapping defined by the enclosing container. For example:

Slice

```
["cpp:type:wstring"]
struct S1
{
    string x;                          // Maps to std::wstring
    ["cpp:type:wstring"] string y;     // Maps to std::wstring
    ["cpp:type:string"] string z;      // Maps to std::string
}

struct S2
{
    string x;                          // Maps to std::string
    ["cpp:type:string"] string y;      // Maps to std::string
    ["cpp:type:wstring"] string z;     // Maps to std::wstring
}
```

With these metadata directives, the strings are mapped as indicated by the comments. By default, narrow strings are encoded as UTF-8, and wide strings use a UTF encoding that is appropriate for the platform on which the application executes. You can override the encoding for narrow and wide strings by registering a string converter with the Ice run time.

String View Mapping in C++

You can use the metadata directive `cpp:view-type:`*`string-view-type`* to map some string parameters to a custom C++ "view-type" of your choice. This view-type can reference memory without owning it, like the experimental `string_view` type. For example:

Slice

```
void sendString(["cpp:view-type:std::experimental::string_view"] string
data);
```

maps to:

C++

```
// Proxy function for synchronous call: input parameter mapped to
string_view type.
void sendString(const std::experimental::string_view&);
```

See Customizing the C++11 Mapping for a detailed description of the `cpp:view-type` metadata directive.

See Also

- Basic Types
- Customizing the C++11 Mapping
- C++11 Strings and Character Encoding

C++11 Mapping for Enumerations

A Slice enumeration maps to the corresponding `enum class` in C++.

For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The generated C++ enumeration is:

C++

```
enum class Fruit : unsigned char { Apple, Pear, Orange };
```

The underlying type is `unsigned char` when the enumeration's largest enumerator value is not greater than 254, otherwise it's the default, `int`.

You can alternatively generate an old-style unscoped enum with the `cpp:unscoped` metadata directive.

Suppose we modify the Slice definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The generated C++ definition now includes an explicit initializer for every enumerator:

C++

```
enum class Fruit : unsigned char { Apple = 0, Pear = 3, Orange = 4 };
```

See Also

- Enumerations
- C++11 Mapping for Structures
- C++11 Mapping for Sequences
- C++11 Mapping for Dictionaries
- Slice Metadata Directives

C++11 Mapping for Structures

A Slice structure maps to a C++ structure.

On this page:

Struct Mapping

Slice structures map to C++ structures with the same name. For each Slice data member, the C++ structure contains a public data member. For example, here is our Employee structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The Slice-to-C++ compiler generates the following definition for this structure:

C++

```
struct Employee
{
    long long int number;
    std::string firstName;
    std::string lastName;

 std::tuple<const long long int&, const ::std::string&, const
::std::string&> ice_tuple() const;
};
```

For each data member in the Slice definition, the C++ structure contains a corresponding public data member of the same name. Constructors are intentionally omitted so that the C++ structure qualifies as a *plain old datatype* (POD).

Comparison Operators

The generated C++ structures use templated comparison operators included from Ice.

C++ Comparison Operators

```
// !=, <, <=, >, >=  are implemented in the same manner
template<class C, typename =
std::enable_if<std::is_member_function_pointer<decltype(&C::ice_tuple)>
::value>>
bool operator==(const C& lhs, const C& rhs)
{
   return lhs.ice_tuple() == rhs.ice_tuple();
}
```

These operators compare a `std::tuple` returned by the generated `ice_tuple()` function.

Generated ice_tuple

```
std::tuple<const long long int&, const ::std::string&, const
::std::string&> ice_tuple() const
{
    return std::tie(number, firstName, lastName);
}
```

Constructors and Assignment Operators

Copy construction and assignment always have deep-copy semantics. You can freely assign structures or structure members to each other without having to worry about memory management. The following code fragment illustrates both comparison and deep-copy semantics:

C++

```
Employee e1, e2;
e1.firstName = "Bjarne";
e1.lastName = "Stroustrup";
e2 = e1;                        // Deep copy
assert(e1 == e2);
e2.firstName = "Andrew";        // Deep copy
e2.lastName = "Koenig";         // Deep copy
assert(e2 < e1);
```

Because strings are mapped to `std::string`, there are no memory management issues in this code and structure assignment and copying work as expected. (The default member-wise copy constructor and assignment operator generated by the C++ compiler do the right thing.)

Default Constructors

Structures have an implicit default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. These default values are mapped to C++ data member initializers.

See Also

- Structures
- C++11 Mapping for Enumerations
- C++11 Mapping for Sequences
- C++11 Mapping for Dictionaries

C++11 Mapping for Sequences

On this page:

Default Sequence Mapping in C++

Here is the definition of our `FruitPlatter` sequence once more:

Slice

```
sequence<Fruit> FruitPlatter;
```

The Slice compiler generates the following definition for the `FruitPlatter` sequence:

C++

```
using FruitPlatter = std::vector<Fruit>;
```

As you can see, the sequence simply maps to a standard `std::vector`, so you can use the sequence like any other vector. For example:

C++

```
// Make a small platter with one Apple and one Orange
//
FruitPlatter p;
p.push_back(Fruit::Apple);
p.push_back(Fruit::Orange);
```

Custom Sequence Mapping in C++

You can override the default mapping of Slice sequences to C++ vectors with the cpp:type and cpp:view-type metadata directives.

For example:

Slice

```
[["cpp:include:list"]]

module Food
{
    enum Fruit { Apple, Pear, Orange }

    ["cpp:type:std::list<Food::Fruit>"]
    sequence<Fruit> FruitPlatter;
}
```

With this metadata directive, the sequence now maps to a C++ `std::list`:

C++

```
#include <list>

namespace Food
{
    using FruitPlatter = std::list<Food::Fruit>;

    // ...
}
```

Custom Mapping for Sequence Parameters

In addition to the default and custom mappings of sequence types as a whole, you can use metadata to customize the mapping of a single operation parameter of type sequence.

The C++ mapping provides a metadata directive for this purpose, `["cpp:array"]`.

Array Mapping for Sequence Parameters in C++

The array mapping for sequence parameters applies only to:

- In parameters, on the client-side and on the server-side
- Out and return parameters provided by the Ice run-time to AMI callbacks
- Out and return parameters provided to marshaled results or AMD callbacks

For example:

Slice

```
interface File
{
    void write(["cpp:array"] Ice::ByteSeq contents);
}
```

The `cpp:array` metadata directive instructs the compiler to map the `contents` parameter to a pair of pointers. With this directive, the `wri`

`te` function on the proxy has the following signature:

C++

```
void write(const std::pair<const Ice::Byte*,
const Ice::Byte*>& contents, const Ice::Context& =
Ice::noExplicitContext);
```

To pass a byte sequence to the server, you pass a pair of pointers; the first pointer points at the beginning of the sequence, and the second pointer points one element past the end of the sequence.

Similarly, for the server side, the `write` method on the skeleton has the following signature:

C++

```
virtual void write(std::pair<const Ice::Byte*, const Ice::Byte*>,
const Ice::Current&) = 0;
```

The passed pointers denote the beginning and end of the sequence as a range `[first, last)` (that is, they use the usual semantics for iterators).

The array mapping is useful to achieve zero-copy passing of sequences. The pointers point directly into the server-side transport buffer; this allows the server-side run time to avoid creating a `vector` to pass to the operation implementation, thereby avoiding both allocating memory for the sequence and copying its contents into that memory.

> You can use the array mapping for any sequence type. However, it provides a performance advantage only for byte sequences (on all platforms) and for sequences of integral or floating point types (on some platforms).

See Also

- Sequences
- Customizing the C++11 Mapping
- C++11 Mapping for Enumerations
- C++11 Mapping for Structures
- C++11 Mapping for Dictionaries
- C++11 Mapping for Operations

C++11 Mapping for Dictionaries

On this page:

Default Dictionary Mapping in C++

Here is the definition of our `EmployeeMap` once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

The following code is generated for this definition:

C++

```
using EmployeeMap = std::map<long long, Employee>;
```

Again, there are no surprises here: a Slice dictionary simply maps to a standard `std::map`. As a result, you can use the dictionary like any other `map`, for example:

C++

```
EmployeeMap em;
Employee e;

e.number = 42;
e.firstName = "Stan";
e.lastName = "Lippman";
em[e.number] = e;

e.number = 77;
e.firstName = "Herb";
e.lastName = "Sutter";
em[e.number] = e;
```

Custom Dictionary Mapping in C++

You can override the default mapping of Slice dictionaries to C++ maps with a `cpp:type` a or `cpp:view-type` metadata directive, for example:

Slice

```
[["cpp:include:unordered_map"]]

["cpp:type:std::unordered_map<long long, Employee>"] dictionary<long,
Employee> EmployeeMap;
```

With this metadata directive, the dictionary now maps to a C++ `std::unordered_map`:

C++

```
#include <unordered_map>

using EmployeeMap = std::unordered_map<long long, Employee>;
```

See Customizing the C++11 Mapping for detailed information about the `cpp:type` and `cpp:view-type` metadata directives.

See Also

- Dictionaries
- Customizing the C++11 Mapping
- C++11 Mapping for Enumerations
- C++11 Mapping for Structures
- C++11 Mapping for Sequences

C++11 Mapping for Constants

Slice constant definitions map to corresponding C++ constant definitions. Slice constants are mapped to `constexpr` constants whenever possible, and to `const` constants otherwise. For example:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

Here are the generated C++ definitions for these constants:

C++

```
constexpr bool          AppendByDefault = true;
constexpr Ice::Byte     LowerNibble =     15;
constexpr std::string   Advice =          "Don't Panic!";
constexpr short         TheAnswer =       42;
constexpr double        PI =              3.1416;

enum class Fruit : unsigned char { Apple, Pear, Orange };
constexpr Fruit         FavoriteFruit =   Fruit::Pear;
```

All constants are initialized directly in the header file, so they are compile-time constants and can be used in contexts where a compile-time constant expression is required, such as to dimension an array or as the `case` label of a `switch` statement.

A Slice string literal that contains non-ASCII characters is mapped by default to a narrow C++ string literal (UTF-8 encoded) with the non-ASCII characters replaced by the corresponding universal character names. For example:

Slice

```
const string Egg = "œuf";
```

is mapped to:

C++

```
constexpr std::string Egg = u8"\u0153uf";
```

If you map a string constant to a `std::wstring`, the non-ASCII characters in the string literal are likewise replaced by universal character names. For example:

Slice

```
const ["cpp:type:wstring"] string LargeEgg = "gros œuf";
```

is mapped to:

C++

```
constexpr std::wstring LargeEgg = L"gros \u0153uf";
```

A Slice string literal that contains universal character names is mapped to a narrow C++ string (UTF-8 encoded) or to a wide C++ string with the universal character names preserved. For example:

Slice

```
const string Heart = "c\u0153ur";
const ["cpp:type:wstring"] string BigHeart = "grand c\u0153ur";
const ["cpp:type:wstring"] string Banana = "\U0001F34C";
```

is mapped to:

C++

```
constexpr std::string Heart = u8"c\u0153ur";
constexpr std::wstring BigHeart = L"grand c\u0153ur";
constexpr std::wstring Banana = L"\U0001F34C";
```

See Also

- Constants and Literals

C++11 Mapping for Exceptions

On this page:

Base Class for Ice Exceptions

The class `Ice::Exception` is the root of the derivation tree for Ice exceptions, and encapsulates functionality that is common to all Ice exceptions:

C++

```
class Exception : public std::exception
{
public:

    Exception();
    Exception(const char* file, int line);

    virtual std::string ice_id() const = 0;
    virtual void ice_print(std::ostream&) const;
    virtual const char* what() const noexcept override;
    virtual void ice_throw() const = 0;

    std::unique_ptr<Exception> ice_clone() const;

    const char* ice_file() const;
    int ice_line() const;
    std::string ice_stackTrace() const;

protected:

    virtual Exception* ice_cloneImpl() const = 0;
};
```

The second constructor stores a file name and line number in the exception that are returned by the `ice_file` and `ice_line` member functions, respectively. This allows you to identify the source of an exception by passing the `__FILE__` and `__LINE__` preprocessor macros to the constructor.

Each exception has the following member functions:

- `ice_clone`
  This member function allows you to polymorphically clone an exception. For example:

C++

```
try
{
    // ...
}
catch(const Ice::UserException& e)
{
   auto copy = e.clone();
}
```

`ice_clone` is useful if you need to make a copy of an exception without knowing its precise run-time type. This allows you to remember the exception and throw it later by calling `ice_throw`. `ice_clone` is implemented through the virtual function `ice_cloneImpl`.

- `ice_id`
  As the name suggests, this member function returns the type ID of the exception. For example, if you call the `ice_id` member function of a `BadZoneName` exception defined in module `M`, it returns the string `"::M::BadZoneName"`. The `ice_id` member function is useful if you catch exceptions generically and want to produce a more meaningful diagnostic, for example:

  C++

  ```
  try
  {
      // ...
  }
  catch(const GenericError& e)
  {
      cerr << "Caught an exception: " << e.ice_id() << endl;
  }
  ```

  If an exception is raised, this code prints the id of the actual exception (such as `::M::BadTimeVal` or `::M::BadZoneName`). For exception that are not defined in Slice, `ice_id` returns the full name of the C++ class.

- `ice_file`
  `ice_file` returns the file name provided to the second constructor of `Exception`.

- `ice_line`
  `ice_line` returns the line number provided to the second constructor of `Exception`.

- `ice_print`
  The default implementation of `ice_print` prints the file name and line number (when available), and the type ID of the exception.

- `ice_stackTrace`
  The `ice_stackTrace` function returns the full stack trace when the exception was constructed, or an empty string, depending on the value of the `Ice.PrintStackTraces` property.

- `ice_throw`
  `ice_throw` allows you to throw an exception without knowing its precise run-time type. It is implemented as:

C++

```
void
GenericError::ice_throw() const
{
    throw *this;
}
```

You can call `ice_throw` to throw an exception that you previously cloned with `ice_clone`.

- `what`
  The default implementation of `what` returns a string created by `ice_print`.

C++ Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

C++

```
class GenericError: public Ice::UserException
{
public:

    GenericError() = default;
    explicit GenericError(const std::string&);

    virtual std::string ice_id() const override;
    virtual void ice_throw() const override;
    std::unique_ptr<GenericError> ice_clone() const;

    std::tuple<const std::string&> ice_tuple() const;

    static const std::string& ice_staticId();

    std::string reason;

protected:
```

```
   virtual Exception* ice_cloneImpl() const override;
};

class BadTimeVal: public GenericError
{
public:

    BadTimeVal() = default;
    explicit BadTimeVal(const std::string&);

    virtual std::string ice_id() const override;
    virtual void ice_throw() const override;
    std::unique_ptr<BadTimeVal> ice_clone() const;

    static const std::string& ice_staticId();

protected:

   virtual Exception* ice_cloneImpl() const override;
};

class BadZoneName: public GenericError
{
public:

    BadZoneName() = default;
    explicit BadZoneName(const std::string&);

    virtual std::string ice_id() const override;
    virtual void ice_throw() const override;
    std::unique_ptr<BadZoneName> ice_clone() const;

    static const std::string& ice_staticId();

protected:
```

```
    virtual Exception* ice_cloneImpl() const override;
};
```

Each Slice exception is mapped to a C++ class with the same name. For each exception member, the corresponding class contains a public data member. (Since `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception has a default constructor. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value. Optional data members are unset unless they declare default values.

An exception also has a second constructor that accepts one argument for each exception member. This constructor allows you to instantiate and initialize an exception in a single statement, instead of having to first instantiate the exception and then assign to its members. For each optional data member, its corresponding constructor parameter uses the same mapping as for operation parameters.

For derived exceptions, the constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

All user exceptions provide an `ice_staticId` static member function that returns the type-id of the exception. For example `BadZoneName::ice_staticId()` returns `"::M::BadZoneName"`.

All user exceptions ultimately inherit from `Ice::UserException`. In turn, `Ice::UserException` inherits from `Ice::Exception`:

C++

```
namespace Ice
{
    class Exception : public std::exception
    {
        // ...
    };
    std::ostream& operator<<(std::ostream&, const Exception&);

    class UserException : public Exception
    {
        // ...
    };
}
```

To make printing more convenient, `operator<<` is overloaded for `Ice::Exception`, so you can also write:

C++

```
try
{
    // ...
}
catch(const Ice::Exception& e)
{
    cerr << e << endl;
}
```

This produces the same output because `operator<<` calls `ice_print` internally. You can optionally provide your own `ice_print` implem entation using the `cpp:ice_print` metadata directive.

For Ice run time exceptions, `ice_print` also shows the file name and line number at which the exception was thrown.

C++ Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice::LocalException` (which, in turn, derives from `Ice::Exception`). `Ice::LocalException` has the usual member functions: `ice_clone`, `ice_id`, `ice_throw`, etc. They also provide an `ice_staticId` static member function like user exceptions.

Recall the inheritance diagram for user and run-time exceptions. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice::Exception`
  This is the root of the complete inheritance tree. Catching `Ice::Exception` catches both user and run-time exceptions. As shown earlier, `Ice::Exception` inherits from `std::exception`.
- `Ice::UserException`
  This is the root exception for all user exceptions. Catching `Ice::UserException` catches all user exceptions (but not run-time exceptions).
- `Ice::LocalException`
  This is the root exception for all run-time exceptions. Catching `Ice::LocalException` catches all run-time exceptions (but not user exceptions).
- `Ice::TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `Ice::ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalExceptio n`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time.
Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- Versioning
- Object Life Cycle

C++11 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Classes and Proxy Handles

On the client side, a Slice interface maps to a class with member functions that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
module M
{
    interface Simple
    {
        void op();
    }
}
```

The Slice compiler generates the following definitions for use by the client:

C++

```
namespace M
{
    class SimplePrx : public virtual Ice::ObjectPrx
    {
    public:
        void op(const Ice::Context& = Ice::noExplicitContext);
        ...
        static const std::string& ice_staticId();
    };
}
```

Your client code interacts directly with the *proxy class*, `M::SimplePrx` in the example above. More generally, the generated proxy class for an interface in module `M` is the C++ proxy class `M::<`*`interface-name`*`>Prx`.

In the client's address space, an instance of the proxy class is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Inheritance from `Ice::Object`

All generated proxy classes inherit directory or indirectly from the `Ice::ObjectPrx` proxy class, reflecting the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

Interface Inheritance

Inheritance relationships among Slice interfaces are maintained in the generated C++ classes. For example:

Slice

```
module M
{
    interface A { ... }
    interface B { ... }
    interface C extends A, B { ... }
}
```

The generated code for `CPrx` reflects the inheritance hierarchy:

C++

```
namespace M
{
    class CPrx : public virtual APrx, public virtual BPrx
    {
        ...
    };
}
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

Receiving Proxies

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. The following code will not compile because the base proxy class for Object has no public constructor:

C++

```
M::SimplePrx s;  // Compile-time error!
```

Proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. When the client receives a proxy from the run time, it is given `std::shared_ptr<proxy class>`.

The client accesses the proxy via this `shared_ptr`; the `shared_ptr` takes care of forwarding operation invocations to its underlying proxy, as well as reference-counting the proxy. This means that no memory-management issues can arise: deallocation of a proxy is automatic and happens once the last `shared_ptr` to the proxy disappears (goes out of scope).

Down-casting Proxies with `checkedCast` and `uncheckedCast`

The Ice namespaces provides two template functions, `checkedCast` and `uncheckedCast`, modeled after `std::dynamic_pointer_cast`:

C++

```
namespace Ice
{
    // Modeled after std::dynamic_pointer_cast. P is the derived proxy
class, for example DerivedPrx.
    template<typename P, typename T, ...>
    std::shared_ptr<P> checkedCast(const std::shared_ptr<T>& b, const
Ice::Context& context = Ice::noExplicitContext)
    {
        ...
    }

    template<typename P, typename T, ...> std::shared_ptr<P>
uncheckedCast(const std::shared_ptr<T>& b)
    {
        ...
    }
}
```

These functions allow you to down-cast a proxy to a more derived proxy type.

Checked cast

A checked cast has the same function for proxies as a C++ `dynamic_cast` has for pointers: it allows you to assign a base proxy to a derived proxy. If the type of the base proxy's target object is compatible with the derived proxy's static type, the assignment succeeds and, after the assignment, the derived proxy denotes the same object as the base proxy. Otherwise, if the type of the base proxy's target object is incompatible with the derived proxy's static type, the derived proxy is set to null. Here is an example to illustrate this:

C++

```
std::shared_ptr<BasePrx> base = ...;     // Initialize base proxy
auto derived = Ice::checkedCast<DerivedPrx>(base); // returns a
shared_ptr<DerivedPrx>
if(derived)
{
    // Base's target object has run-time type Derived,
    // use derived...
}
else
{
    // Base has some other, unrelated type
}
```

The expression `checkedCast<DerivedPrx>(base)` tests whether `base` points at an object of type `Derived` (or an object with a type that is derived from `Derived`). If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to null.

> A `checkedCast` results in a remote message, `ice_isA`, to the server. The message effectively asks the server "is the object denoted by this reference of type Derived?"

The reply from the server is communicated to the application code in the form of a successful (non-null) or unsuccessful (null) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of a remote Ice object is without confirmation from the server. (For example, the server may replace the implementation of the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive a `ConnectFailedException;` if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ObjectNotExistException`.

Unchecked cast

In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast. An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object, for example:

C++

```
std::shared_ptr<BasePrx> base = ...;     // Initialize to point at a De
rived
auto derived = Ice::uncheckedCast<DerivedPrx>(base);
// Use derived...
```

You should use an `uncheckedCast` only if you are certain that target object indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not checked in any way; it does not contact the object in the server and, if it fails, it does not return null. If you use the proxy resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `OperationNotExistException`, but, depending on the circumstances, the Ice run time may also report an exception indicating that unmarshaling has failed, or even silently return garbage results.

> Calling `uncheckedCast` on a proxy that is already of the desired proxy type returns immediately that proxy. Otherwise, `uncheckedCast` creates a new instance of the desired proxy class.

Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization, it is common to receive a proxy of static type `Ice::ObjectPrx`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

Typed Proxy Factory Methods in C++

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

C++11

```
std::shared_ptr<Ice::ObjectPrx> proxy =
communicator->stringToProxy(...);
proxy = proxy->ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, the corresponding C++ factory member functions return a proxy of the same type as the current proxy, therefore it is generally not necessary to down-cast after calling such a factory. The example below demonstrates these semantics:

C++

```
auto base = communicator->stringToProxy(...);
auto hello = checkedCast<HelloPrx>(base);
hello = hello->ice_invocationTimeout(10000); // Type is preserved
hello->sayHello();
```

The only exceptions are the factory member functions `ice_facet` and `ice_identity`. Calls to either of these functions may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in C++

You can compare proxies for equality. By default, proxy comparison compares all aspects of a proxy, including the object identity, facet name, addressing information, and all the proxy settings; two proxies compare equal only if they are identical in all respects. The mapping provides helper functions to simplify the comparison of proxies stored in `shared_ptr` values.

Note however that the more common use case is determining whether two proxies denote the same Ice object, in which case you should only be comparing their object identities. To compare the object identities of two proxies, you can use helper functions and classes in the `Ice` namespace:

C++

```
namespace Ice
{
    bool proxyIdentityLess(const std::shared_ptr<ObjectPrx>&,
const std::shared_ptr<ObjectPrx>&);
    bool proxyIdentityEqual(const std::shared_ptr<ObjectPrx>&,
const std::shared_ptr<ObjectPrx>&);
    bool proxyIdentityAndFacetLess(const std::shared_ptr<ObjectPrx>&,
const std::shared_ptr<ObjectPrx>&);
    bool proxyIdentityAndFacetEqual(const std::shared_ptr<ObjectPrx>&,
const std::shared_ptr<ObjectPrx>&);

    struct ProxyIdentityLess : std::binary_function<bool,
std::shared_ptr<ObjectPrx>&, std::shared_ptr<ObjectPrx>&>
    {
        bool operator()(const std::shared_ptr<ObjectPrx>& lhs, const
std::shared_ptr<ObjectPrx>& rhs) const
        {
            return proxyIdentityLess(lhs, rhs);
        }
    };

    struct ProxyIdentityEqual ...
    struct ProxyIdentityAndFacetLess ...
    struct ProxyIdentityAndFacetEqual ...
}
```

The `proxyIdentityEqual` function returns true if the object identities embedded in two proxies are the same and ignores other information in the proxies, such as facet and transport information. To include the facet name in the comparison, use `proxyIdentityAndF`

acetEqual instead.

The proxyIdentityLess function establishes a total ordering on proxies. It is provided mainly so you can use object identity comparison with STL sorted containers. (The function uses name as the major ordering criterion, and category as the minor ordering criterion.) The proxyIdentityAndFacetLess function behaves similarly to proxyIdentityLess, except that it also compares the facet names of the proxies when their identities are equal.

proxyIdentityEqual and proxyIdentityAndFacetLess allow you to correctly compare proxies for object identity. The example below demonstrates how to use proxyIdentityEqual:

C++

```
std::shared_ptr<Ice::ObjectPrx> p1 = ...;        // Get a proxy...
std::shared_ptr<Ice::ObjectPrx> p2 = ...;        // Get another proxy...

if(!Ice::proxyIdentityEqual(p1, p2)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- C++11 Mapping for Operations
- Example of a File System Client in C++11
- Versioning

C++11 Mapping for Operations

On this page:

Basic C++ Mapping for Operations

As we saw in the C++ mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The proxy class for the `Node` interface, tidied up to remove irrelevant detail, is as follows:

C++

```
namespace Filesystem
{
    class NodePrx : public virtual Ice::ObjectPrx
    {
    public:
        std::string name(const Ice::Context& = Ice::noExplicitContext);
        // ...
    };
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

C++

```
shared_ptr<NodePrx> node = ...; // Initialize proxy
string name = node->name();     // Get name via RPC
```

This code calls `name` on the proxy class instance, which sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

Because the return value is of type `string`, it is safe to ignore the return value. For example, the following code contains no memory leak:

C++

```
shared_ptr<NodePrx> node = ...; // Initialize proxy
node->name();                   // Useless, but no leak
```

This is true for all mapped Slice types: you can safely ignore the return value of an operation, no matter what its type — return values are always returned by value. If you ignore the return value, no memory leak occurs because the destructor of the returned value takes care of deallocating memory as needed.

Normal and `idempotent` Operations in C++

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy methods is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent string op2();
    idempotent void op3(string s);
}
```

The proxy class for this interface looks like this:

C++

```
class ExamplePrx : public virtual Ice::ObjectPrx
{
public:
    std::string op1(const Ice::Context& = Ice::noExplicitContext);
    std::string op2(const Ice::Context& =
Ice::noExplicitContext);                // idempotent
    void op3(const std::string&, const Ice::Context& =
Ice::noExplicitContext);   // idempotent
    // ...
};
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the `idempotent`

keyword.

Passing Parameters in C++

In-Parameters in C++

The parameter passing rules for the C++ mapping are very simple: parameters are passed either by value (for small values) or by `const` reference (for values that are larger than a machine word). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following code for this definition:

C++

```
struct NumberAndString
{
    int x;
    std::string str;
    // ...
};

using StringSeq = std::vector<std::string>;
using StringTable = std::map<long long int, StringSeq>;

class ClientToServer : public virtual Ice::ObjectPrx
{
public:
    void op1(int, float, bool, const std::string&, const Ice::Context& =
Ice::noExplicitContext);
    void op2(const NumberAndString&, const StringSeq&,
const StringTable&, const Ice::Context& = Ice::noExplicitContext);
    void op3(const ClientToServerPrx&, const Ice::Context& =
Ice::noExplicitContext);
    // ...
};
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

C++

```
shared_ptr<ClientToServerPrx> p = ...;   // Get proxy...

p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14;
bool b = true;
string s = "Hello world!";
p->op1(i, f, b, s);                     // Pass simple variables

NumberAndString ns = { 42, "The Answer" };
StringSeq ss;
ss.push_back("Hello world!");
StringTable st;
st[0] = ss;
p->op2(ns, ss, st);                     // Pass complex variables

p->op3(p);                              // Pass proxy
```

You can pass either literals or variables to the various operations. Because everything is passed by value or `const` reference, there are no

memory-management issues to consider.

Out-Parameters in C++

The C++ mapping passes out-parameters by reference. Here is the Slice definition once more, modified to pass all parameters in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns, out StringSeq ss,
out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Slice compiler generates the following code for this definition:

C++

```
class ServerToClient : public virtual Ice::ObjectPrx
{
public:
    void op1(int&, float&, bool&, std::string&, const Ice::Context& =
Ice::noExplicitContext);
    void op2(NumberAndString&, StringSeq&, StringTable&, const
Ice::Context& = Ice::noExplicitContext);
    void op3(ServerToClientPrx&, const Ice::Context& =
Ice::noExplicitContext);
    // ...
};
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

C++

```
shared_ptr<ServerToClientPrx> p = ...;      // Get proxy...

int i;
float f;
bool b;
string s;

p->op1(i, f, b, s);
// i, f, b, and s contain updated values now

NumberAndString ns;
StringSeq ss;
StringTable st;

p->op2(ns, ss, st);
// ns, ss, and st contain updated values now

p->op3(p);
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes variables to an operation; once the operation completes, the values of those variables will be set by the server.

It is worth having another look at the final call:

C++

```
p->op3(p);      // Weird, but well-defined
```

Here, p is the proxy that is used to dispatch the call. That same variable p is also passed as an out-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe: the Ice run time will correctly handle this situation.

Optional Parameters in C++

The mapping for optional parameters is the same as for required parameters, except each optional parameter is encapsulated in an Ice::optional value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

The C++ mapping for this operation is shown below:

C++

```
Ice::optional<int> execute(const Ice::optional<std::string>& params,
Ice::optional<float>& value, ...);
```

For an optional output parameter, the Ice run time resets the client's `optional` instance if the server does not supply a value for the parameter, therefore it is safe to pass an `optional` instance that already has a value.

A well-behaved program must not assume that an optional parameter always has a value.

Chained Invocations in C++

Consider the following simple interface containing two operations, one to set a value and one to get it:

Slice

```
interface Name
{
    string getName();
    void setName(string name);
}
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

C++

```
p2->setName(p1->getName());
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`. There are no memory-management or exception safety issues with this code.

Exception Handling in C++

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as C++ exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

C++

```
auto child = ...;               // Get Child proxy handle...
try
{
    child->askToCleanUp();      // Give it a try...
}
catch(const Tantrum& t)
{
    cout << "The child says: " << t.reason << endl;
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

C++

```
int
main(int argc, char* argv[])
{
    int status = 1;
    try
    {
        auto child = ...;           // Get proxy handle...
        try
        {
            child->askToCleanUp();  // Give it a try...
            child->praise();        // Give positive feedback...
        }
        catch(const Tantrum& t)
        {
            cout << "The child says: " << t.reason << endl;
            child->scold();         // Recover from error...
        }
        status = 0;
    }
    catch(const Ice::LocalException& e)
    {
        cerr << "Unexpected run-time error: " << e << endl;
    }
    // ...
    return status;
}
```

For efficiency reasons, you should always catch exceptions by `const` reference. This permits the compiler to avoid calling the exception's copy constructor (and, of course, prevents the exception from being sliced to a base type).

Exceptions and Out-Parameters in C++

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may have still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

> This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

Exceptions and Return Values in C++

For return values, C++ provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown. (Of course, this guarantee holds only if you do not use the same variable as both an out-parameter and to receive the return value of an invocation).

See Also

- Operations
- Slice for a Simple File System
- C++11 Mapping for Interfaces

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.

C++11 Mapping for Optional Values

Slice optional data members and parameters are mapped to `Ice::optional<T>` values.

The template `Ice::optional` is a placeholder for the future C++17 `std::optional`, and provides the same functions, with identical semantics. Please refer to `std::optional` for a full description of this template class and its associated functions, types and objects:

C++

```
// When std::optional is available, Ice::optional and the associated
functions, types and objects will be defined as follows:

namespace Ice
{
    template<class T> using optional = std::optional<T>;

    using std::operator==;
    using std::operator!=;
    using std::operator<;
    using std::operator<=;
    using std:operator>;
    using std::operator>=;

    using std::make_optional;
    using std::swap;
    using nullopt_t = std::nullopt_t;
    using std::nullopt;
    using bad_optional_access = std::bad_optional_access;
    using in_place_t = std::in_place_t;
    using std::in_place;
}
```

See Also

- Optional Data Members
- Operations
- C++11 Mapping for Operations

C++11 Mapping for Classes

On this page:

Basic C++ Mapping for Classes

A Slice class is mapped to a C++ class with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions). Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

The Slice compiler generates the following code for this definition:

C++

```
class TimeOfDay : public Ice::Value
{
public:
    short hour;
    short minute;
    short second;
    std::string tz;

    TimeOfDay() = default;
    TimeOfDay(short, short, short, string);

    virtual std::shared_ptr<Ice::Value> ice_clone() const;

 std::tuple<const short&, const short&, const short&, const
::std::string&> ice_tuple() const;

 static const std::string& ice_staticId();

protected:
    // ...
};
```

There are a number of things to note about this generated code:

1. The generated class `TimeOfDay` inherits from `Ice::Value`. This means that all classes implicitly inherit from `Ice::Value` which is the ultimate ancestor of all classes.
2. The generated class contains a public member for each Slice data member.
3. The generated class has a constructor that takes one argument for each data member, as well as a default constructor.
4. The generated class has a function, `ice_tuple`, which returns a `std::tuple` of the class' data members. Ice uses this tuple to perform `TimeOfDay` class comparison.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice::Value`

Classes implicitly inherit from a common base class, `Value`, which is mapped to `Ice::Value` in C++.

`Ice::Value` is a very simple base class with just a few member functions:

C++

```
namespace Ice
{
    class Value
    {
    public:
        virtual ~Value() = default;

        virtual void ice_preMarshal();
        virtual void ice_postUnmarshal();

        virtual std::shared_ptr<Value> ice_clone() const;

  virtual std::shared_ptr<SlicedData> ice_getSlicedData() const;

        static const std::string& ice_staticId();
        ...
    };
}
```

The member functions of `Ice::Value` behave as follows:

- `ice_preMarshal`
  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_clone`
  This function returns a `shared_ptr` that holds a shallow copy of this value.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `nullptr` otherwise.
- `ice_staticId`
  This function returns the static type ID of a class.

Class Data Members in C++

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the

generated class contains a corresponding public data member. Optional data members are mapped to instances of the `Ice::optional` template.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    ["protected"] string tz;    // GMT, EST etc.
}
```

The Slice compiler produces the following generated code for this definition:

C++

```
class TimeOfDay : public Ice::Value
{
public:
    // ctors etc.

protected:

    short hour;
    short minute;
    short second;
    std::string tz;
};
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;
}
```

Class Constructors with C++

Classes have several constructors:

- a default constructor that default-constructs each data member
  This default constructor is no-op and implemented as `= default`. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.
  To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition, and the Slice compiler will generate data member initializers for the corresponding C++ data members. Optional data members are unset unless they declare default values.
- a constructor with one parameter for each data member (the *one-shot* constructor)
  This constructor allows you to construct and initialize a class instance in a single statement. For each optional data member, its corresponding constructor parameter uses the same mapping as for operation parameters, allowing you to pass its initial value or `Ice::nullopt` to indicate an unset value. Each parameter is passed by value: each movable parameter is moved into the corresponding data member, while other parameters are copied.
- a copy constructor
- a move constructor

For derived classes, the one-shot constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order. For example:

Slice

```
class Base
{
    int i;
}

class Derived extends Base
{
    string s;
    string greeting = "hello";
}
```

This generates:

C++

```
class Base : public Ice::Value
{
public:
    int i;

    Base() = default;
    explicit Base(int) { ... }
    Base(const Base&);
    Base(Base&&);
    // ...
};

class Derived : public Base
{
public:
    std::string s;
    std::string greeting = "hello";

    Derived() = default;
    Derived(int, string, string); // one-shot ctor
    Derived(const Derived&&);
 Derived(Derived&&);

    // ...
};
```

Note that single-parameter constructors are defined as `explicit`, to prevent implicit argument conversions.

Class with Operations in C++

> **Deprecated Feature**
> Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

Operations on classes are not mapped at all into the corresponding C++ class. The generated C++ class is the same whether the Slice class has operations or not, except for the nested Result structs described later in this section.

The Slice to C++ compiler also generates a separate <*class-name*>`Disp` class, which can be used to implement an Ice object with these operations. For example:

Slice

```
class FormattedTimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;
    string format();
}
```

results in the following generated code:

C++

```
class FormattedTimeOfDay : public Ice::Value
{
     // ... operation format() not mapped at all here
};

// Disp class for servant implementation
class FormattedTimeOfDayDisp : public virtual Ice::Object
{
public:
     virtual std::string format(const Ice::Current& =
Ice::noExplicitCurrent) = 0;
     // ...
};
```

This Disp class is the *C++ skeleton class* for this Slice class. Skeleton classes are described in the Server-Side C++98 Mapping for Interfaces.

Like skeleton classes for interfaces, the generated C++ Disp classes always derive from their base class(es) virtually. This virtual inheritance allows the reuse of implementation-classes through multiple inheritance.

If an operation has a return value and one or more out parameters, or no return value and two or more out parameters, the generated C++ class provides a nested C++ struct used for asynchronous calls. For example:

Slice

```
class XYZ
{
    int x;
    string op(out int y);
}
```

results in the following generated code:

C++

```
class XYZ : public Ice::Value
{
public:

    int x;
    struct OpResult
    {
       string returnValue;
       int y;
    };
};

// Disp class for servant implementation
class XYZDisp : public virtual Ice::Object
{
public:
     virtual std::string op(int&, const Ice::Current& =
Ice::noExplicitCurrent) = 0;
     // ...
};
```

Value Factories in C++

Value factories may be used for classes with or without operations and are *not* deprecated.

Value factories allow you to create classes derived from the C++ class generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
interface Time
{
    TimeOfDay get();
}
```

The Ice run time will by default create and return a plain `TimeOfDay` instance.

If you wish, you can create your own custom derived class, and tell Ice to create and return these instances instead. For example:

C++

```
class CustomTimeOfDay : public TimeOfDay
{
public:
    std::string format() { ... prints formatted data members ... }
};
```

You then create and register a value factory for your custom class with your Ice communicator:

C++

```
auto communicator = ...;
communicator->getValueFactoryManager()->add(
    [](const string& type)
    {
        assert(type == TimeOfDay::ice_staticId());
        return new CustomTimeOfDay;
    },
    TimeOfDay::ice_staticId()));
```

See Also

- Classes
- C++11 Mapping for Operations
- C++11 Mapping for Optional Values
- Asynchronous Method Invocation (AMI) in C++11
- Dispatch Interceptors
- Value Factories

Asynchronous Method Invocation (AMI) in C++11

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

Callback and Future-Based APIs

Each Slice operation is mapped to two `<operation-name>Async` functions on the corresponding proxy class:

- a *future-based* function, that returns a `std::future`; this future object delivers the operation's return value and out parameters
- a *callback-based* function, that take callbacks as `std::function` parameters; this is the full featured and somewhat lower-level function

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Besides the synchronous proxy functions, `slice2cpp` generates the following asynchronous proxy functions:

C++

```
// Future-based function
// with the P = std::promise (the default), it's equivalent to:
// std::future<std::string> getNameAsync(int number, const Ice::Context&
context = Ice::noExplicitContext);
//
template<template<typename> class P = std::promise>
auto getNameAsync(int number, const Ice::Context& context =
Ice::noExplicitContext)
        -> decltype(std::declval<P<std::string>>().get_future());

// Callback-based function
//
std::function<void()>
getNameAsync(int number,
             std::function<void(std::string)> response,
             std::function<void(std::exception_ptr)> exception =
nullptr,
             std::function<void(bool)> sent = nullptr,
             const Ice::Context& context = Ice::noExplicitContext);
```

Future-Based Async Function

The future-based async function returns a `std::future` object. It can also return a custom future object if you specify the associated promise template. For example:

C++

```
auto e = ... // get an Employees proxy
auto fut1 = e->getNameAsync(99); // fut1 is a plain std::future, created
from a std::promise
cout << "Employee name is: " << fut1.get() << endl; // fut1.get() blocks
until the result is available

auto fut2 = e->getNameAsync<std::experimental::promise>(98); // fut2 is
a std::experimental::future, created from the provided promise template
```

The future's result depends on the operation's parameters:

- when the operation has no return value or out parameter, the result type is `void`.
- when the operation has a return value, or no return value but a single out parameter, the result is this return value or out parameter.
- when the operation has a return value and one or more out parameters (or no return value and two or more out parameters), the result is a generated struct *`<operation-name>Result`* (with the first letter capitalized) in the mapped interface class (or in the main mapped class for a class with operations). This struct has public data members named after the operation parameters; the data member for the return value is named `returnValue`.

For example, if we add a new out parameter to `getName`:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number, out string email);
    }
}
```

The Slice to C++ compiler will generate:

C++

```
class Employees : public virtual Ice::Object
{
public:

    struct GetNameResult
    {
        std::string returnValue;
        std::string email;
    };
    ...
};

class EmployeesPrx : public virtual Ice::ObjectPrx
{
public:

    template<template<typename> class P = std::promise>
    auto getNameAsync(int number, const Ice::Context& ctx =
Ice::noExplicitContext)
            ->
decltype(std::declval<P<Employees:GetNameResult>>().get_future());

    ...
};
```

You would typically use `auto` to avoid typing the name of this `Result` struct:

C++11

```
auto e = ... // get an Employees proxy
auto fut = e->getNameAsync(99); // get future<Employees::GetNameResult>
```

Callback-Based Async Function

With the callback-based async function, you must provide all the mandatory in-parameters of the operation, followed by a response callback. You can then optionally provide an exception callback and a sent callback.

These callbacks are described below:

- response callback
  The Ice run time calls the response callback to deliver asynchronously the response from a two-way invocation that completes successfully. The signature for this response callback is `std::function<void(`*`return-type, first-out-type, second-out-type...`*`)>`. The response callback for an operation with no return or out parameter has no parameters. Otherwise, all the parameters to this callback function are passed by value, to allow your callback to adopt (move) the memory allocated by the Ice run time (the caller).

- exception callback
  The Ice run time calls the exception callback (when provided) to deliver asynchronously the result of an invocation that completes with an error. This exception callback accepts a single `std::exception_ptr` parameter, passed by value, that can hold any type of exception.

- sent callback
  When you call an `Async` function the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. The Ice run time calls the sent callback (if provided) to notify you that the request has been accepted by the transport. `sent` accepts a single `bool` parameter, set to `true` when the request is sent synchronously, and `false` otherwise.

For example:

C++

```
auto e = ... // get an Employees proxy
e->getNameAsync(99,
                [](string name) { cout << "Employee name is: " << name
<< endl; },
                [](exception_ptr eptr)
                {
                    try
                    {
                        rethrow_exception(eptr);
                    }
                    catch(const std::exception& ex)
                    {
                        cerr << "Request failed: " << ex.what() << endl;
                    }
                });
```

The Ice run time calls these callbacks using a thread from the communicator's client thread pool, with one exception: the sent callback is called by the thread making the invocation when the request is sent synchronously.

Asynchronous Exception Semantics

If an invocation raises an exception, the exception is reported by the exception callback or by the future, even if the actual error condition for the exception was encountered during the call to the `Async` function ("on the way out"). The advantage of this behavior is that all exception handling is located in the same place (instead of being present twice, once where you call the `Async` function, and again where you retrieve the result) .

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `Async` function throws `CommunicatorDestroyedException`.
  This is necessary because, once the communicator is destroyed, its client thread pool is no longer available.

- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

Asynchronous Oneway Invocations

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an `Async` function on a oneway proxy for an operation that returns values or raises a user exception, the `Async` function throws `TwowayOnlyException`.

An `async` oneway invocation does not call the response callback with the callback API; you use the sent callback to make sure the invocation was successfully sent. With the future-based API, the returned future is a `future<void>` and this future is made ready when the invocation is sent.

Canceling an Asynchronous Invocation

The `Async` function with callback parameters returns a cancel function-object (a `std::function<void()>`). You can use this function-object to cancel the invocation, for example:

C++

```
auto e = ... // get an Employees proxy
auto cancel = e->getNameAsync(99, [](string name) { cout << "Employee
name is: " << name << endl; });
cancel(); // no longer interested in this name
```

Calling this cancel function-object prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. This cancelation is purely local and has no effect on the server.

Canceling an invocation that has already completed has no effect. Otherwise, a canceled invocation is considered to be completed, meaning the exception callback (if provided) receives an `Ice::InvocationCanceledException`.

Polling for Completion

The future-based `Async` function allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

C++

```
FileHandle file = open(...);
shared_ptr<FileTransferPrx> ft = ...;
const int chunkSize = ...;

int offset = 0;
while(!file.eof())
{
    ByteSeq bs;
    bs = file.read(chunkSize); // Read a chunk
    ft->send(offset, bs);      // Send the chunk
    offset += bs.size();
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

C++

```
FileHandle file = open(...);
shared_ptr<FileTransferPrx> ft = ...;
const int chunkSize = ...;
int offset = 0;

deque<future<void>> results;
const int numRequests = 5;

while(!file.eof())
{
    ByteSeq bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    auto fut = ft->sendAsync(offset, bs);
    offset += bs.size();

    results.push_back(std::move(fut));

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.size() > numRequests)
    {
        results.front().get();
        results.pop_front();
    }
}

// Wait for any remaining requests to complete.
while(!results.empty())
{
    results.front().get();
    results.pop_front();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Flow Control

Asynchronous method invocations never block the thread that calls the `Async` function : the Ice run time checks to see whether it can write

the request to the local transport. If it can, it does so immediately in the caller's thread. Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background.

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The callback API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

For example:

C++

```
auto e = ...; // get an Employees proxy

e->getNameAsync(99,
                [](string name) { ... handle name ... },
                [](exception_ptr ex) { ... handle exception ... },
                [](bool) { ... increase sent counter ... });
```

Asynchronous Batch Requests

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous proxy method on a batch oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

A batch oneway invocation never calls the response or sent callbacks with the callback API. With the future-based API, the returned future for a batch oneway invocation is always ready and indicates the successful queuing of the batch invocation. The future completes exceptionally if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides `Async` versions of this function so you can flush batch requests asynchronously:

C++

```
// Future-based function
// with the P = std::promise (the default), it's equivalent to:
// std::future<void> ice_flushBatchRequestsAsync();
//
template<template<typename> class P = std::promise>
auto ice_flushBatchRequestsAsync() ->
decltype(std::declval<P<void>>().get_future());

// Callback-based function
//
std::function<void()>
ice_flushBatchRequestsAsync(std::function<void(std::exception_ptr)> ex,
std::function<void(bool)> sent = nullptr);
```

The bool value returned by the future-based function indicates whether the flush was performed synchronously (return value is `true`) or asynchronously (return value is `false`).

Similar `flushBatchRequestsAsync` functions are also available on `Communicator` and `Connection`:

C++

```
// Future-based function
// with the P = std::promise (the default), it's equivalent to:
// std::future<void> flushBatchRequestsAsync(Ice::CompressBatch
compress);
//
template<template<typename> class P = std::promise>
auto flushBatchRequestsAsync(Ice::CompressBatch compress) ->
decltype(std::declval<P<void>>().get_future())

// Callback-based function
//
std::function<void()>
flushBatchRequestsAsync(Ice::CompressBatch compress,
                        std::function<void(std::exception_ptr)>
exception,
                        std::function<void(bool)> sent = nullptr);
```

As described on the Batched Invocations page, `flushBatchRequests` on `Communicator` and `Connection` flushes only requests made with fixed proxies.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch

Using Slice Checksums in C++11

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2cpp`, the `--checksum` option causes the compiler to generate code in each C++ source file that accumulates checksums in a global map. A copy of this map can be obtained by calling a function defined in the header file `Ice/SliceChecksums.h`:

C++

```
namespace Ice
{
    Ice::SliceChecksumDict sliceChecksums();
}
```

In order to verify a server's checksums, a client could simply compare the dictionaries using the equality operator. However, this is not feasible if it is possible that the server might be linked with more Slice definitions than the client. A more general solution is to iterate over the local checksums as demonstrated below:

C++

```
Ice::SliceChecksumDict serverChecksums = ...
auto localChecksums = Ice::sliceChecksums();

for(const auto& p : localChecksums)
{
    auto q = serverChecksums.find(p.first);
    if(q == serverChecksums.end())
    {
        // No match found for type id!
    }
    else if(p.second != q->second)
    {
        // Checksum mismatch!
    }
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums
- slice2cpp Command-Line Options

Example of a File System Client in C++11

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The C++ code shown here hardly differs from the code you would write for an ordinary C++ program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local C++ object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side C++ mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

C++

```
#include <Ice/Ice.h>
#include <Filesystem.h>
#include <iostream>
#include <iterator>
```

```
#include <stdexcept>

using namespace std;
using namespace Filesystem;

static void
listRecursive(const shared_ptr<DirectoryPrx>& dir, int depth = 0)
{
    // ...
}

int
main(int argc, char* argv[])
{
    try
    {
        // Create Ice communicator
        //
        Ice::CommunicatorHolder ich(argc, argv);

        // Create a proxy for the root directory
        //
        auto base = ich->stringToProxy("RootDir:default -p 10000");
        if(!base)
        {
            throw std::runtime_error("Could not create proxy");
        }

        // Down-cast the proxy to a Directory proxy
        //
        auto rootDir = Ice::checkedCast<DirectoryPrx>(base);
        if(!rootDir)
        {
            throw std::runtime_error("Invalid proxy");
        }

        // Recursively list the contents of the root directory
        //
        cout << "Contents of root directory:" << endl;
        listRecursive(rootDir);
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        return 1;
    }
```

```
    return 0;
}
```

1. The code includes a few header files:
   - `Ice/Ice.h`:
     Always included in both client and server source files, provides definitions that are necessary for accessing the Ice run time.
   - `Filesystem.h`:
     The header that is generated by the Slice compiler from the Slice definitions in `Filesystem.ice`.
   - `iostream`:
     The client uses the `iostream` library to produce its output.
   - `iterator`:
     The implementation of `listRecursive` uses an STL iterator.
   - `stdexcept`:
     `main` can throw `std::runtime_error`, which is declared in this header file.
2. The code adds `using` declarations for the `std` and `Filesystem` namespaces.
3. The structure of the code in `main` follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
4. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`:

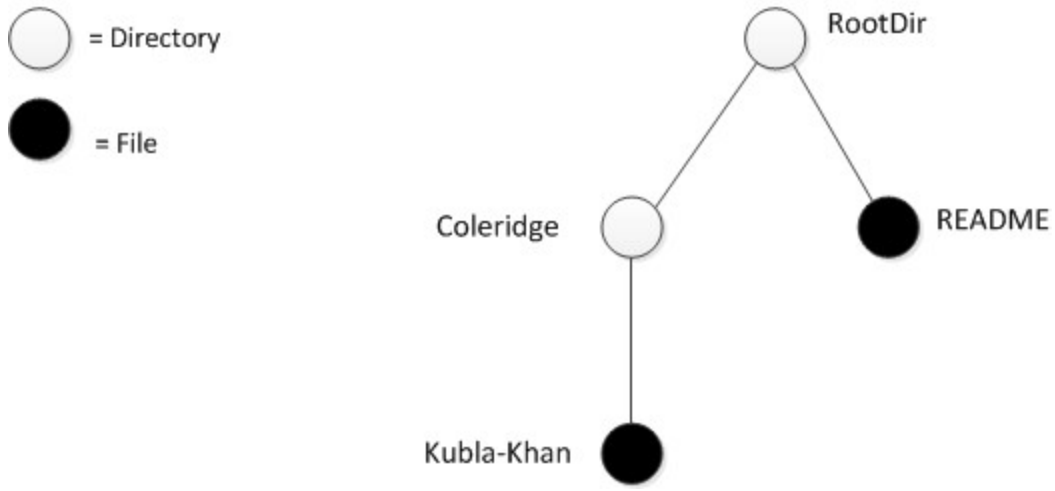
C++

```
// Recursively print the contents of directory "dir" in
// tree fashion. For files, show the contents of each file.
// The "depth" parameter is the current nesting level
// (for indentation).

static void
listRecursive(const shared_ptr<DirectoryPrx>& dir, int depth = 0)
{
    string indent(++depth, '\t');
    NodeSeq contents = dir->list();
    for(const auto& node : contents)
    {
        auto subdir = Ice::checkedCast<DirectoryPrx>(node);
        cout << indent << node->name() << (subdir ? " (directory):" : "
(file):") << endl;
        if(subdir)
        {
            listRecursive(subdir, depth);
        }
        else
        {
            auto file = Ice::uncheckedCast<FilePrx>(node);
            auto text = file->read();
            for(const auto& line : text)
            {
                cout << indent << "\t" << line << endl;
            }
        }
    }
}
```

The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the list operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy. This cast succeeds if the node is a directory, and fails (returns a null proxy) if it's not.
2. The code prints the name of the file or directory and then, depending on the result of the previous cast, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Example of a File System Server in C++11
- Object Life Cycle
- IceGrid

Server-Side Slice-to-C++11 Mapping

The mapping for Slice data types to C++11 is identical on the client side and server side. This means that everything in Client-Side Slice-to-C++11 Mapping also applies to the server side. However, for the server side, there are a few additional things you need to know — specifically how to:

- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time.

Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Although the examples we present are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any C++11 mapping rules beyond those we describe here.

Topics

Server-Side C++11 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing virtual functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in C++

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interf ace:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The Slice compiler generates the following definition for this interface:

C++

```
namespace Filesystem
{
    class Node : public virtual Ice::Object
    {
    public:
        virtual std::string name(const Ice::Current&) = 0;
        // ...
    };
    // ...
}
```

For the moment, we will ignore a number of other member functions of this class. The important points to note are:

- As for the client side, Slice modules are mapped to C++ namespaces with the same name, so the skeleton class definition is nested in the namespace `Filesystem`.
- The name of the skeleton class is the same as the name of the Slice interface (`Node`).
- The skeleton class contains a pure virtual member function for each operation in the Slice interface.
- The skeleton class is an abstract base class because its member functions are pure virtual.
- The skeleton class inherits from `Ice::Object` (which forms the root of the Ice object hierarchy).

`Ice::Object` Base Class for C++ Servants

`Object` is mapped to the `Ice::Object` class in C++:

C++

```
namespace Ice
{
    class Object
    {
    public:
        virtual ~Object() = default;

        virtual bool ice_isA(std::string, const Current&) const;
        virtual void ice_ping(const Current&) const;
        virtual std::vector< std::string> ice_ids(const Current&) const;
        virtual std::string ice_id(const Current&) const;

        static const std::string& ice_staticId();

        virtual bool ice_dispatch(Ice::Request&,
                                  std::function<bool()> = nullptr,

std::function<bool(std::exception_ptr)> = nullptr);

        struct Ice_invokeResult
        {
            bool returnValue;
            std::vector<Byte> outParams;
        };

         ...
    };
}
```

The member functions of `Ice::Object` behave as follows:

- `ice_isA`
  This function returns `true` if target object implements the given type ID, and `false` otherwise.
- `ice_ping`
  `ice_ping` provides a basic reachability test for the servant.
- `ice_ids`
  This function returns a string sequence representing all of the type IDs implemented by this servant, including `::Ice::Object`.
- `ice_id`
  This function returns the type ID of the most-derived interface implemented by this servant.
- `ice_staticId`
  This static function returns the type ID of the target class: `::Ice::Object` when called on `Ice::Object`.
- `ice_dispatch`
  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.

The nested struct `Ice_invokeResult` supplies the result for calls to `ice_invokeAsync` on proxies.

Servant Classes in C++

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

C++

```
#include <Filesystem.h> // Slice-generated header

class NodeI : public virtual Filesystem::Node
{
public:
    NodeI(const std::string&);
    virtual std::string name(const Ice::Current&) override;
private:
    std::string _name;
};
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.)

Note that `NodeI` inherits from `Filesystem::Node`, that is, it derives from its skeleton class. It is a good idea to always use virtual inheritance when defining servant classes. Strictly speaking, virtual inheritance is necessary only for servants that implement interfaces that use multiple inheritance; however, the `virtual` keyword does no harm and, if you add multiple inheritance to an interface hierarchy half-way through development, you do not have to go back and add a `virtual` keyword to all your servant classes.

As far as Ice is concerned, the `NodeI` class must implement only a single member function: the pure virtual `name` function that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. Obviously, the constructor initializes the `_name` member and the `name` function returns its value:

C++

```
NodeI::NodeI(const std::string& name) : _name(name)
{
}

std::string
NodeI::name(const Ice::Current&) const
{
    return _name;
}
```

Normal and `idempotent` Operations in C++

The `name` member function of the `NodeI` skeleton is not a `const` member function. However, given that the operation does not modify the state of its object, it really should be a `const` member function. We can achieve this by adding the `["cpp:const"]` metadata directive. For example:

Slice

```
interface Example
{
                void normalOp();

    idempotent  void idempotentOp();

    ["cpp:const"]
    idempotent  void readonlyOp();
}
```

The skeleton class for this interface looks like this:

C++

```
class Example : public virtual Ice::Object
{
public:
    virtual void normalOp(const Ice::Current&) = 0;
    virtual void idempotentOp(const Ice::Current&) = 0;
    virtual void readonlyOp(const Ice::Current&) const = 0;
    // ...
};
```

Note that `readonlyOp` is mapped as a `const` member function due to the `["cpp:const"]` metadata directive; normal and `idempotent` operations (without the metadata directive) are mapped as ordinary, non-`const` member functions.

See Also

- Slice for a Simple File System
- C++11 Mapping for Interfaces
- Parameter Passing in C++11
- Raising Exceptions in C++11

Parameter Passing in C++11

Parameter passing on the server side generally follows the same rules as for the client side. Additionally, every operation receives a trailing parameter of type `Ice::Current`. For example, the `name` operation of the `Node` interface has no parameters, but the corresponding `name` method of the servant interface has a single parameter of type `Current`. We will ignore this parameter for now.

On this page:

Server-Side Mapping for Parameters in C++11

For each parameter of a Slice operation, the C++ mapping generates a corresponding parameter for the virtual member function in the skeleton. Parameter passing on the server side follows these rules:

- in-parameters are passed by value only
- out-parameters are passed by reference
- return values are passed by value
- optional parameters are enclosed in `Ice::optional` values

> Compared to the client side rules, the only difference is for in-parameters: on the client side, they are mapped to value or const reference (depending on the parameter type), while on the server side they are always passed by value.
>
> On the client side, you allocate these in-parameters and Ice only needs to read them, so const reference is fine for parameters like strings and vectors. On the server side, Ice allocates these parameters and then relinquishes them to your servant: getting these parameters by value allows your servant to adopt (move) them.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
module M
{
    interface Example
    {
        string op(string sin, out string sout);
    }
}
```

The generated skeleton class for this interface looks as follows:

C++

```
namespace M
{
    class Example : public virtual Ice::Object
    {
    public:
        virtual std::string op(string, std::string&,
const Ice::Current&) = 0;
        // ...
    };
}
```

As you can see, there are no surprises here. For example, we could implement op as follows:

C++

```
std::string
ExampleI::op(std::string sin, std::string& sout, const Ice::Current&)
{
    cout << sin << endl;        // In parameters are initialized
    sout = "Hello World!";      // Assign out parameter
    return "Done";              // Return a string
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal C++ rules and do not require special-purpose API calls or memory management.

Thread-Safe Marshaling in C++

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For C++ applications, this can affect servant methods that return instances of Slice classes or types referencing Slice classes.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following Slice:

Slice

```
sequence<int> IntSeq;
sequence<IntSeq> IntIntSeq;
sequence<string> StringSeq;
class Grid
{
    StringSeq xLabels;
    StringSeq yLabels;
    IntIntSeq values;
}

interface GridIntf
{
   Grid getGrid();
   void clearValues();
}
```

And the following servant implementation:

C++

```
class GridIntfI : public GridIntf
{
public:
    std::shared_ptr<Grid> getGrid(const Ice::Current&);
    void clear(const Ice::Current&);

private:
    std::mutex _mutex;
    std::shared_ptr<Grid> _grid;
};

std::shared_ptr<Grid>
GridIntfI::getGrid(const Ice::Current&)
{
    std::lock_guard<std::mutex> lock(_mutex);
    return _grid;
}

void
GridIntfI::clearValues(const Ice::Current&)
{
    std::lock_guard<std::mutex> lock(_mutex);
    _grid->values.clear();
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned class in preparation to send a reply

message, it is possible for another thread to dispatch the `clearValues` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `clearValues` operations does not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `clearValues` replaces `_grid` with a copy that contains empty values, leaving the previous contents of `_grid` unchanged:

C++

```
void GridIntfI::clearValues(const Ice::Current&)
{
    std::lock_guard<std::mutex> lock(_mutex);
    shared_ptr<Grid> grid = make_shared<Grid>();
    grid->xLabels = _grid->xLabels;
    grid->yLabels = _grid->yLabels;
    _grid = grid;
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the `values` array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `marshaled-result` metadata directive changes the signature of the corresponding servant method, if that operation returns mutable types. The metadata directive has the following effects:

- For an operation `op` that returns one or multiple values and at least one of those values has a mutable type, the Slice compiler generates an `OpMarshaledResult` class and the return type of the servant method becomes `OpMarshaledResult`.
- The constructor for `OpMarshaledResult` takes an extra argument of type `Current`. The servant must supply the `Current` in order for the results to be marshaled correctly.

The metadata directive has no effect on the proxy mapping, nor does it affect the servant mapping of Slice operations that return `void` or return only immutable values.

> You can also annotate an interface with the `marshaled-result` metadata and it will be applied to all of the interface's operations.

After applying the metadata, we can now implement the `Grid` servant as follows:

C++

```
GetGridMarshaledResult
GridIntfI::getGrid(const Ice::Current& currrent)
{
    return GetGridMarshaledResult(_grid, curr); // _grid is marshaled
immediately
}
```

Here are more examples to demonstrate the mapping:

Slice

```
class C { ... }
struct S { ... }
sequence<string> Seq;

interface Example
{
    C getC();

    ["marshaled-result"]
    C getC2();

    void getS(out S val);

    ["marshaled-result"]
    void getS2(out S val);

    string getValues(string name, out Seq val);

    ["marshaled-result"]
    string getValues2(string name, out Seq val);
}
```

Review the generated code below to see the changes that the presence of the metadata causes in the servant method signatures:

C++

```
class Example : public virtual Ice::Object
{
public:
    class GetCMarshaledResult : public Ice::MarshaledResult
    {
    public:
        GetCMarshaledResult(const std::shared_ptr<C>&, const
Ice::Current&);
    };

    class GetS2MarshaledResult : public Ice::MarshaledResult
    {
    public:
        GetS2MarshaledResult(const S&, const Ice::Current&);
    };

    class GetValues2MarshaledResult : public Ice::MarshaledResult
    {
    public:
        GetValues2MarshaledResult(const std::string&, const Seq&, const
Ice::Current&);
    };

    virtual std::shared_ptr<C> getC(const Ice::Current&) = 0;
    virtual GetC2MarshaledResult getC2(const Ice::Current&) = 0;
    virtual S getS(const Ice::Current&) = 0;
    virtual GetS2MarshaledResult getS2(const Ice::Current&) = 0;
    virtual std::string getValues(std::string, Seq&, const
Ice::Current&) = 0;
    virtual GetValues2MarshaledResult getValues2(std::string, const
Ice::Current&) = 0;
};
```

See Also

- Server-Side C++11 Mapping for Interfaces
- C++11 Mapping for Operations
- C++11 Mapping for Optional Values
- Raising Exceptions in C++11
- The Current Object

Raising Exceptions in C++11

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

C++

```
void
Filesystem::FileI::write(Filesystem::Lines text, const Ice::Current&)
{
    // Try to write the file contents here...
    // Assume we are out of space...
    if(error)
    {
        throw Filesystem::GenericError("file too large");
    }
}
```

No memory management issues arise in the presence of exceptions.

If you throw an arbitrary C++ exception (such as an `int` or other unexpected type), the Ice run time catches the exception and then returns an `UnknownException` to the client.

The server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw a run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

See Also

- Run-Time Exceptions
- C++11 Mapping for Exceptions
- Server-Side C++11 Mapping for Interfaces
- Parameter Passing in C++11

Object Incarnation in C++11

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must follow the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a C++ Servant

Instantiating a servant means to allocate an instance on the heap:

C++

```
auto servant = make_shared<NodeI>("Fred");
```

This code creates a new `NodeI` instance and `shared_ptr` to hold it.

Creating an Identity in C++

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the

`category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

C++

```
Ice::Identity id;
id.name = "Fred"; // Not unique, but good enough for now
```

Activating a C++ Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

C++

```
_adapter->add(servant, id);
```

Note the two arguments to `add`: the smart pointer to the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant pointer from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

Servant Life Time

Putting the preceding points together, we can write a simple function that instantiates and activates one of our `NodeI` servants. For this example, we use a simple helper function called `activateServant` that creates and activates a servant with a given identity:

C++

```
void
activateServant(const string& name)
{
    auto servant = make_shared<NodeI>(name); // Refcount == 1
    Ice::Identity id;
    id.name = name;
    _adapter->add(servant, id);  // Refcount == 2
}   // Refcount == 1
```

Note that we create the servant on the heap and that, once `activateServant` returns, we lose the last remaining handle to the servant (because the `servant` variable goes out of scope). The question is, what happens to the heap-allocated servant instance? The answer lies in the `shared_ptr` semantics:

- When the new servant is instantiated, the reference count of its newly created `shared_ptr` is initially 1.

- Calling `add` passes the servant's `shared_ptr` to the object adapter which keeps a copy internally. This increments the reference count of the ptr to 2.
- When `activateServant` returns, the destructor of the `servant` variable decrements the reference count of the ptr to 1.

The net effect is that the servant's `shared_ptr` and the servant itself are retained on the heap for as long as the servant is in the servant map of its object adapter. (If we deactivate the servant, that is, remove it from the servant map, the reference count drops to zero and the memory occupied by the servant and its `shared_ptr` are reclaimed; we discuss these life cycle issues in Object Life Cycle.)

UUIDs as Identities in C++

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. Ice provides a helper function to create such identities, `generateUUID`:

C++

```
#include <Ice/Ice.h>
#include <iostream>

using namespace std;

int
main()
{
    cout << Ice::generateUUID() << endl;
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can rewrite the code shown earlier like this:

C++

```
void
activateServant(const string& name)
{
    auto servant = make_shared<NodeI>(name);
    _adapter->addWithUUID(servant);
}
```

Creating Proxies in C++

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Hello World Application. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in C++

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

C++

```
auto servant = make_shared<NodeI>(name); // servant is a
shared_ptr<NodeI>
auto proxy = uncheckedCast<NodePrx>(_adapter->addWithUUID(servant)); //
proxy is a shared_ptr<NodePrx>

// Pass proxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice::ObjectPrx`.

Direct Proxy Creation in C++

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

C++

```
Ice::Identity id;
id.name = Ice::generateUUID();
auto o = _adapter->createProxy(id); // o is a shared_ptr<Ice::ObjectPrx>
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Hello World Application
- Object Adapter States
- Servant Locators
- Object Life Cycle

Asynchronous Method Dispatch (AMD) in C++11

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool. If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run time.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

On this page:

Enabling AMD with Metadata in C++

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

Slice

```
["amd"] interface I
{
    bool isValid();
    float computeRate();
}

interface J
{
    ["amd"] void startProcess();
    int endProcess();
}
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface or class level) minimizes the amount of generated code and, more importantly, minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

AMD Mapping in C++

The `["amd"]` metadata changes the name of the dispatch pure virtual function to `<operation-name>Async`. This dispatch function returns `void` and accepts the operation's in-parameters by value, followed by two callback parameters provided by the Ice run time:

- `std::function<void(param1, param2...)>`
  This callback allows the server to report the successful completion of the operation. If the operation has a non-`void` return type, the first parameter is the return value. Parameters corresponding to the operation's `out` parameters follow the return value, in the order of declaration. These parameters are passed by value or by const reference depending on the type of the parameter.
- `std::function<void(std::exception_ptr)>`
  This allows the server to raise any standard C++ exception, Ice run-time exception, or Ice user exception.

For example, suppose we have defined the following operation:

Slice

```
interface I
{
    ["amd"] string foo(short s, out long l);
}
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

C++

```
void fooAsync(short, std::function<void(const std::string&, long long)>,
std::function<void(std::exception_ptr)>,
              const Ice::Current&) = 0;
```

The AMD `Async` function looks like very much the AMI `Async` function with callbacks, but these functions are not identical. The table below highlights their differences:

	AMI	AMD
In parameters	Passed by value or by const reference, depending on the parameter type	Passed by value
Return value and out parameters	Passed by value to the response callback	Passed by value or by const reference, depending on the parameter type
Callback functions	3 callbacks: response, exception and sent	2 callbacks: response and exception
Last parameter	`Ice::Context`	`Ice::Current`

AMD Exceptions in C++

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).

These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

Although we recommend that the exception callback be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run-time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the exception callback. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the exception callback, local exceptions may undergo translation.

AMD Example in C++

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

Slice

```
module Demo
{
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {}

    interface Model
    {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    }
}
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo::Model` and supplies a definition for the `interpolateAsync` method:

C++

```
class ModelI : public Demo::Model
{
public:
    virtual void interpolateAsync(Demo::Grid, float,
                                  std::function<void(const
Demo::Grid&)>,

std::function<void(std::exception_ptr)>,
                                  const Ice::Current&);

private:
    std::deque<Job> _jobs;
    std::mutex _mutex;
};
```

The implementation of `interpolateAsync` uses synchronization to safely record the callback functions and arguments in a `Job` that is added to a queue:

C++

```
void
ModelI::interpolateAsync(Demo::Grid data, float factor,
                         std::function<void(const Demo::Grid&)>
response,
                         std::function<void(std::exception_ptr)>
exception,
                         const Ice::Current&)
{
    std::lock_guard<std::mutex> lock(_mutex);
    _jobs.emplace_back(std::move(data), factor, std::move(response),
std::move(exception));
}
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute` to perform the interpolation. `Job` is defined as follows:

C++

```
class Job
{
public:

    Job(Demo::Grid&&, float,
        std::function<void(const Demo::Grid&)>&&,
        std::function<void(std::exception_ptr)>&&);

    void execute();

private:

    void interpolateGrid(); // can throw RangeError

    Demo::Grid _data;
    const float _factor;
    const std::function<void(const Demo::Grid&)> _response;
    const std::function<void(std::exception_ptr)> _exception;
};
```

The implementation of `execute` uses `interpolateGrid` (not shown) to perform the computational work:

C++

```
void Job::execute()
{
    try
    {
        interpolateGrid();
        _response(_data);
    }
    catch(...)
    {
        _exception(std::current_exception());
    }
}
```

If `interpolateGrid` throws an exception such as range error, we capture this exception and pass it to the `_exception` callback. If the interpolation was successful, `_response` is called to send the modified grid back to the client.

See Also

- Asynchronous Method Invocation (AMI) in C++11
- The Ice Threading Model
- User Exceptions
- Run-Time Exceptions

Example of a File System Server in C++11

This page presents the source code for a C++ server that implements our file system and communicates with the client we wrote earlier. The code is fully functional.

The server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

On this page:

Implementing a File System Server in C++

We have now seen enough of the server-side C++ mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of two source files:

- `Server.cpp`
  This file contains the server main program.
- `FilesystemI.cpp`
  This file contains the implementation for the file system servants.

Server `main` Program in C++

Our server main program, in the file `Server.cpp`, consists of two functions, `main` and `run`. `main` creates and destroys an Ice communicator, and `run` uses this communicator instantiate our file system objects:

C++

```
#include <Ice/Ice.h>
#include <FilesystemI.h>

using namespace std;
using namespace Filesystem;

int run(const shared_ptr<Ice::Communicator>& communicator);

int main(int argc, char* argv[])
{
    int status = 0;

    try
    {
        //
        // CtrlCHandler must be created before the communicator or any
other threads are started
        //
        Ice::CtrlCHandler ctrlCHandler;
```

```
        //
        // CommunicatorHolder's ctor initializes an Ice communicator,
        // and its dtor destroys this communicator.
        //
        Ice::CommunicatorHolder ich(argc, argv);
        auto communicator = ich.communicator();

        auto appName = argv[0];
        ctrlCHandler.setCallback(
            [communicator, appName](int)
            {
                communicator->shutdown();
                cerr << appName << ": received signal, shutting down" <<
endl;
            });

        //
        // The communicator initialization removes all Ice-related
arguments from argc/argv
        //
        if(argc > 1)
        {
            cerr << argv[0] << ": too many arguments" << endl;
            status = 1;
        }
        else
        {
            status = run(communicator);
        }
    }
    catch(const std::exception& ex)
    {
        cerr << ex.what() << endl;
        status = 1;
    }

    return status;
}

int
run(const shared_ptr<Ice::Communicator>& communicator)
{
    //
    // Create an object adapter.
    //
    auto adapter = communicator->createObjectAdapterWithEndpoints(
        "SimpleFilesystem", "default -h localhost -p 10000");

    //
```

```
    // Create the root directory (with name "/" and no parent)
    //
    auto root = make_shared<DirectoryI>("/", nullptr);
    root->activate(adapter);

    //
    // Create a file called "README" in the root directory
    //
    auto file = make_shared<FileI>("README", root);
    auto text = Lines({"This file system contains a collection of
poetry."});
    file->write(text, Ice::emptyCurrent);
    file->activate(adapter);

    //
    // Create a directory called "Coleridge" in the root directory
    //
    auto coleridge = make_shared<DirectoryI>("Coleridge", root);
    coleridge->activate(adapter);

    //
    // Create a file called "Kubla_Khan" in the Coleridge directory
    //
    file = make_shared<FileI>("Kubla_Khan", coleridge);
    text =
        {
            "In Xanadu did Kubla Khan",
            "A stately pleasure-dome decree:",
            "Where Alph, the sacred river, ran",
            "Through caverns measureless to man",
            "Down to a sunless sea."
        };
    file->write(text, Ice::emptyCurrent);
    file->activate(adapter);

    //
    // All objects are created, allow client requests now
    //
    adapter->activate();

    //
    // Wait until we are done
    //
    communicator->waitForShutdown();
```

```
    return 0;
}
```

There is quite a bit of code here, so let us examine each section in detail:

C++

```
#include <Ice/Ice.h>
#include <FilesystemI.h>
using namespace std;
using namespace Filesystem;
```

The code includes the header file `FilesystemI.h`. That file includes the header file that is generated by the Slice compiler, `Filesystem.h`.

Two `using` declarations, for the namespaces `std` and `Filesystem`, permit us to be a little less verbose in the source code.

The next part of the source code is the `main` function:

C++

```
//
        // CtrlCHandler must be created before the communicator or any
other threads are started
        //
        Ice::CtrlCHandler ctrlCHandler;

        //
        // CommunicatorHolder's ctor initializes an Ice communicator,
        // and its dtor destroys this communicator.
        //
        Ice::CommunicatorHolder ich(argc, argv);
        auto communicator = ich.communicator();

        auto appName = argv[0];
        ctrlCHandler.setCallback(
            [communicator, appName](int)
            {
                communicator->shutdown();
                cerr << appName << ": received signal, shutting down" <<
endl;
            });
```

We create a CtrlCHandler object which allows us to catch CTRL-C and similar signals in a portable fashion in C++. When the server receives such a signal, it shuts down the communicator (as shown above). Shutting down the communicator in turn makes `waitForShutdown` return .

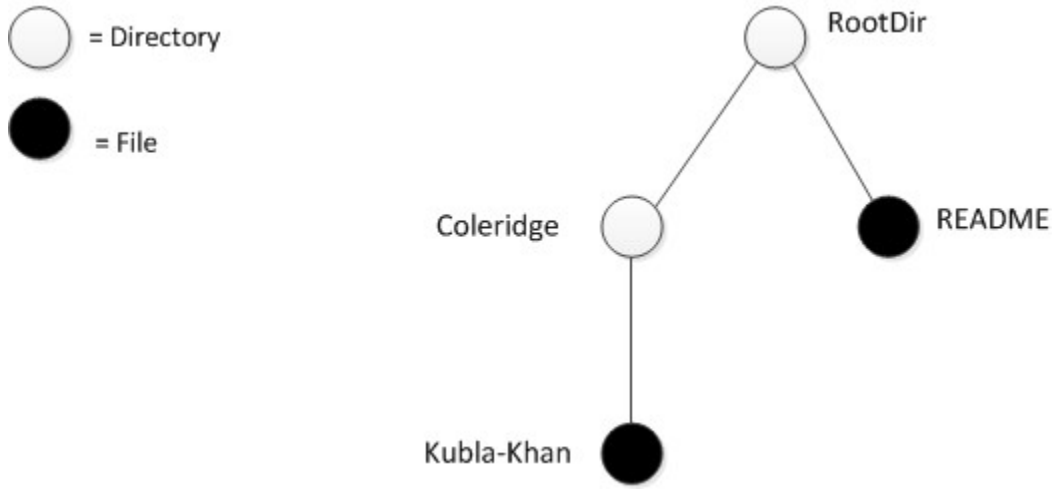
`main` later calls `run`, which is typically blocked on `waitForShutdown`:

C++

```
int
run(const shared_ptr<Ice::Communicator>& communicator)
{
    //
    // Create an object adapter.
    //
    auto adapter = communicator->createObjectAdapterWithEndpoints(
        "SimpleFilesystem", "default -h localhost -p 10000");

    ...


    //
    // All objects are created, allow client requests now
    //
    adapter->activate();

    //
    // Wait until we are done
    //
    communicator->waitForShutdown();

    return 0;
}
```

Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either

type of servant accepts two parameters: the name of the directory or file to be created, and a shared pointer to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

C++

```
auto root = make_shared<DirectoryI>("/", nullptr);
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in the illustration above.

C++

```
//
        // Create the root directory (with name "/" and no parent)
        //
        auto root = make_shared<DirectoryI>("/", nullptr);
        root->activate(adapter);

        //
        // Create a file called "README" in the root directory
        //
        auto file = make_shared<FileI>("README", root);
        auto text = Lines({"This file system contains a collection of
poetry."});
        file->write(text, Ice::emptyCurrent);
        file->activate(adapter);

        //
        // Create a directory called "Coleridge" in the root directory
        //
        auto coleridge = make_shared<DirectoryI>("Coleridge", root);
        coleridge->activate(adapter);

        //
        // Create a file called "Kubla_Khan" in the Coleridge directory
        //
        file = make_shared<FileI>("Kubla_Khan", coleridge);
        text =
        {
            "In Xanadu did Kubla Khan",
            "A stately pleasure-dome decree:",
            "Where Alph, the sacred river, ran",
            "Through caverns measureless to man",
            "Down to a sunless sea."
        };
        file->write(text, Ice::emptyCurrent);
        file->activate(adapter);
```

We first create the root directory and a file `README` within the root directory. (Note that we pass the shared pointer to the root directory as the

parent pointer when we create the new node of type `FileI`.)

After creating each servant, the code calls `activate` on the servant. (We will see the definition of this member function shortly.) The `activate` member function adds the servant to the ASM.

The next step is to fill the file with text:

C++

```
auto file = make_shared<FileI>("README", root);
        auto text = Lines({"This file system contains a collection of
poetry."});
        file->write(text, Ice::emptyCurrent);
        file->activate(adapter);
```

Recall that Slice sequences map to C++ vectors. The Slice type `Lines` is a sequence of strings, so the C++ type `Lines` is a vector of strings; we add a line of text to our `README` file by calling `push_back` on that vector.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

C++

```
        file->write(text, Ice::emptyCurrent);
```

This statement is interesting: the server code invokes an operation on one of its own servants. The Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary C++ function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the above illustration.

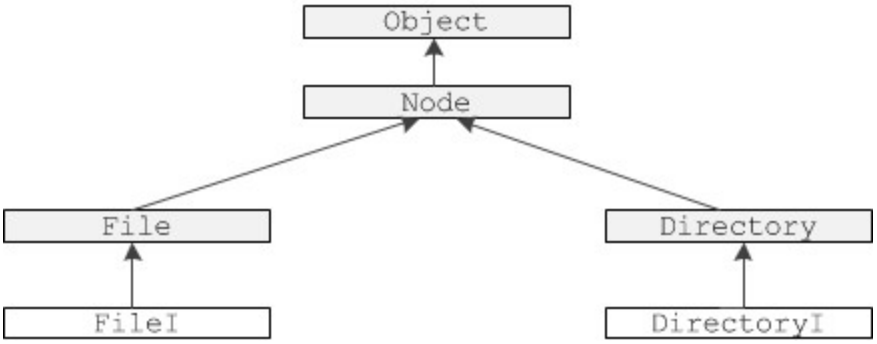
Servant Class Definitions in C++

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the C++ classes `FileI` and `DirectoryI`. This means that our servant classes might look as follows:

C++

```
namespace Filesystem
{
    class FileI : public File
    {
       // ...
    };

    class DirectoryI : public Directory
    {
       // ...
    };
}
```

This leads to the C++ class structure as shown:



File system servants using interface inheritance.

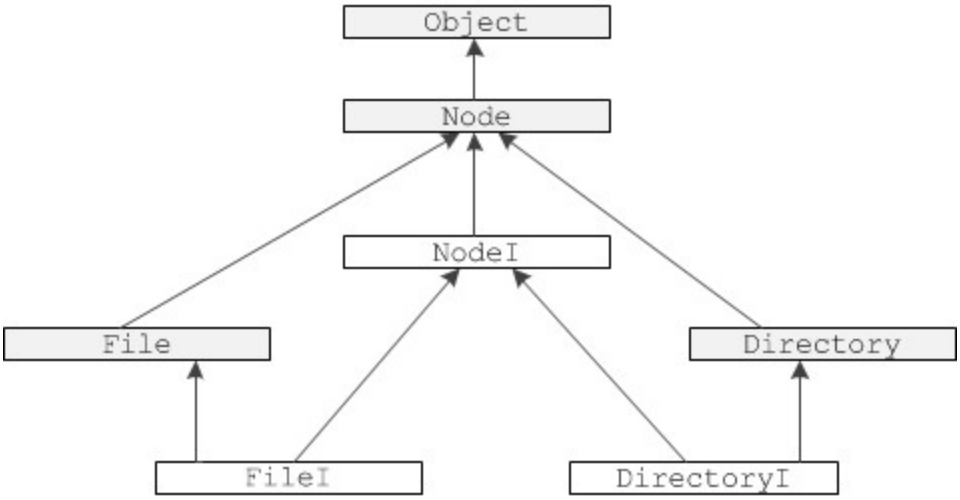
The shaded classes in the illustration above are skeleton classes and the unshaded classes are our servant implementations. If we implement our servants like this, `FileI` must implement the pure virtual operations it inherits from the `File` skeleton (`read` and `write`), as well as the operation it inherits from the `Node` skeleton (`name`). Similarly, `DirectoryI` must implement the pure virtual function it inherits from the `Directory` skeleton (`list`), as well as the operation it inherits from the `Node` skeleton (`name`). Implementing the servants in this way uses interface inheritance from `Node` because no implementation code is inherited from that class.

Alternatively, we can implement our servants using the following definitions:

C++

```
namespace Filesystem
{
    class NodeI : public virtual Node
    {
      // ...
    };

    class FileI : public File, public NodeI
    {
      // ...
    };

    class DirectoryI : public Directory, public NodeI
    {
      // ...
    };
}
```

This leads to the C++ class structure shown:



File system servants using implementation inheritance.

In this implementation, `NodeI` is a concrete base class that implements the `name` operation it inherits from the `Node` skeleton. `FileI` and `DirectoryI` use multiple inheritance from `NodeI` and their respective skeletons, that is, `FileI` and `DirectoryI` use implementation inheritance from their `NodeI` base class.

Either implementation approach is equally valid. Which one to choose simply depends on whether we want to re-use common code provided by `NodeI`. For the implementation that follows, we have chosen the second approach, using implementation inheritance.

Given the structure in the above illustration and the operations we have defined in the Slice definition for our file system, we can add these operations to the class definition for our servants:

C++

```
namespace Filesystem
{
    class NodeI : public virtual Node
    {
    public:
        virtual std::string name(const Ice::Current&) override;
    };

    class FileI : public File, public NodeI
    {
    public:
        virtual Lines read(const Ice::Current&) override;
        virtual void write(const Lines&, const Ice::Current&) override;
    };

    class DirectoryI : public Directory, public NodeI
    {
    public:
        virtual NodeSeq list(const Ice::Current&) override;
    };
}
```

This simply adds signatures for the operation implementations to each class. Note that the signatures must exactly match the operation signatures in the generated skeleton classes — if they do not match exactly, you end up overloading the pure virtual function in the base

class instead of overriding it, meaning that the servant class cannot be instantiated because it will still be abstract. To avoid signature mismatches, you can copy the signatures from the generated header file (`Filesystem.h`), or you can use the `--impl` option with `slice2cpp` to generate header and implementation files that you can add your application code to.

Now that we have the basic structure in place, we need to think about other methods and data members we need to support our servant implementation. Typically, each servant class hides the copy constructor and assignment operator, and has a constructor to provide initial state for its data members. Given that all nodes in our file system have both a name and a parent directory, this suggests that the `NodeI` class should implement the functionality relating to tracking the name of each node, as well as the parent-child relationships:

C++

```
namespace Filesystem
{
    class NodeI : public virtual Node, public
std::enable_shared_from_this<NodeI>
    {
    public:
        virtual std::string name(const Ice::Current&) override;
        NodeI(const std::string&, const std::shared_ptr<DirectoryI>&);
        void activate(const std::shared_ptr<Ice::ObjectAdapter>&);

    private:
        std::string _name;
        Ice::Identity _id;
        std::shared_ptr<DirectoryI> _parent;
    };
}
```

The `NodeI` class has a private data member to store its name (of type `std::string`) and its parent directory (of type `shared_ptr<DirectoryI>`). The constructor accepts parameters that set the value of these data members. For the root directory, by convention, we pass a `nullptr` to the constructor to indicate that the root directory has no parent. The `activate` member function adds the servant to the ASM (which requires access to the object adapter) and connects the child to its parent.

The `FileI` servant class must store the contents of its file, so it requires a data member for this. We can conveniently use the generated `Lines` type (which is a `std::vector<std::string>`) to hold the file contents, one string for each line. Because `FileI` inherits from `NodeI`, it also requires a constructor that accepts the file name, and parent directory, leading to the following class definition:

C++

```
namespace Filesystem
{
    class FileI : public File, public NodeI
    {
    public:
        virtual Lines read(const Ice::Current&) override;
        virtual void write(const Lines&, const Ice::Current&) override;
        FileI(const std::string&, const std::shared_ptr<DirectoryI>&);
    private:
        Lines _lines;
    };
}
```

For directories, each directory must store its list of child notes. We can conveniently use the generated `NodeSeq` type (which is a `vector<s`

hared_ptr<NodePrx>>) to do this. Because `DirectoryI` inherits from `NodeI`, we need to add a constructor to initialize the directory name and its parent directory. As we will see shortly, we also need a private helper function, `addChild`, to make it easier to connect a newly created directory to its parent. This leads to the following class definition:

C++

```
namespace Filesystem
{
    class DirectoryI : public Directory, public NodeI
    {
    public:
        virtual NodeSeq list(const Ice::Current&) const override;
        DirectoryI(const std::string&,
const std::shared_ptr<DirectoryI>&);
        void addChild(const std::shared_ptr<NodePrx>& child);
    private:
        NodeSeq _contents;
    };
}
```

Servant Header File Example

Putting all this together, we end up with a servant header file, `FilesystemI.h`, as follows:

C++

```
#include <Ice/Ice.h>
#include <Filesystem.h>

namespace Filesystem
{
    class DirectoryI;

    class NodeI : public virtual Node, public
std::enable_shared_from_this<NodeI>
    {
    public:
        virtual std::string name(const Ice::Current&) override;
        NodeI(const std::string&, const std::shared_ptr<DirectoryI>&);
        void activate(const std::shared_ptr<Ice::ObjectAdapter>&);
    private:
        std::string _name;
        Ice::Identity _id;
        std::shared_ptr<DirectoryI> _parent;
    };

    class FileI : public File, public NodeI
    {
    public:
        virtual Lines read(const Ice::Current&) override;
        virtual void write(Lines, const Ice::Current&) override;
        FileI(const std::string&, const std::shared_ptr<DirectoryI>&);
    private:
        Lines _lines;
    };

    class DirectoryI : public Directory, public NodeI
    {
    public:
        virtual NodeSeq list(const Ice::Current&);
        DirectoryI(const std::string&, const
std::shared_ptr<DirectoryI>&);
        void addChild(const std::shared_ptr<NodePrx>&);
    private:
        NodeSeq _contents;
    };
}
```

The Servant Implementation in C++

The implementation of our servants is mostly trivial, following from the class definitions in our `FilesystemI.h` header file.

Implementing `FileI`

The implementation of the `read` and `write` operations for files is trivial: we simply store the passed file contents in the `_lines` data member. The constructor is equally trivial, simply passing its arguments through to the `NodeI` base class constructor:

C++

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

void
Filesystem::FileI::write(Lines text, const Ice::Current&)
{
    _lines = std::move(text);
}

Filesystem::FileI::FileI(const string& name,
const shared_ptr<DirectoryI>& parent)
    : NodeI(name, parent)
{
}
```

Implementing `DirectoryI`

The implementation of `DirectoryI` is equally trivial: the `list` operation simply returns the `_contents` data member and the constructor passes its arguments through to the `NodeI` base class constructor:

C++

```
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current&)
{
    return _contents;
}

Filesystem::DirectoryI::DirectoryI(const string& name,
const shared_ptr<DirectoryI>& parent)
    : NodeI(name, parent)
{
}

void
Filesystem::DirectoryI::addChild(const shared_ptr<NodePrx>& child)
{
    _contents.push_back(child);
}
```

The only noteworthy thing is the implementation of `addChild`: when a new directory or file is created, the constructor of the `NodeI` base class calls `addChild` on its own parent, passing it the proxy to the newly-created child. The implementation of `addChild` appends the passed reference to the contents list of the directory it is invoked on (which is the parent directory).

Implementing `NodeI`

The name operation of our `NodeI` class is again trivial: it simply returns the `_name` data member:

C++

```
std::string
Filesystem::NodeI::name(const Ice::Current&)
{
    return _name;
}
```

The `NodeI` constructor creates an identity for the servant:

C++

```
Filesystem::NodeI::NodeI(const string& name,
const shared_ptr<DirectoryI>& parent)
    : _name(name), _parent(parent)
{
    _id.name = parent ? Ice::generateUUID() : "RootDir";
}
```

For the root directory, we use the fixed identity `"RootDir"`. This allows the client to create a proxy for the root directory. For directories other than the root directory, we use a UUID as the identity.

Finally, `NodeI` provides the `activate` member function that adds the servant to the ASM and connects the child node to its parent directory:

C++

```
void
Filesystem::NodeI::activate(const shared_ptr<Ice::ObjectAdapter>& adapt
er)
{
    auto self =
Ice::uncheckedCast<NodePrx>(adapter->add(shared_from_this(), _id));
    if(_parent)
    {
        _parent->addChild(self);
    }
}
```

This completes our servant implementation. The complete source code is shown here once more:

C++

```
#include <Ice/Ice.h>
#include <FilesystemI.h>
using namespace std;

// Slice Node::name() operation
string
Filesystem::NodeI::name(const Ice::Current&)
{
    return _name;
}

// NodeI constructor
Filesystem::NodeI::NodeI(const string& name, const
shared_ptr<DirectoryI>& parent)
    : _name(name),
      _parent(parent)
{
    // Create an identity. The root directory has the fixed identity
"RootDir"
    _id.name = parent ? Ice::generateUUID() : "RootDir";
}

// NodeI activate() member function
void
Filesystem::NodeI::activate(const shared_ptr<Ice::ObjectAdapter>&
adapter)
{
    auto self =
Ice::uncheckedCast<NodePrx>(adapter->add(shared_from_this(), _id));
    if(_parent)
    {
        _parent->addChild(self);
    }
}

// Slice File::read() operation
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

// Slice File::write() operation
void
Filesystem::FileI::write(Filesystem::Lines text, const Ice::Current&)
{
    _lines = std::move(text);
}

// FileI constructor
```

```
Filesystem::FileI::FileI(const string& name, const
shared_ptr<DirectoryI>& parent)
    : NodeI(name, parent)
{
}

// Slice Directory::list() operation
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current&)
{
    return _contents;
}


// DirectoryI constructor
Filesystem::DirectoryI::DirectoryI(const string& name, const
shared_ptr<DirectoryI>& parent)
    : NodeI(name, parent)
{
}

// addChild is called by the child in order to add
// itself to the _contents member of the parent
void
Filesystem::DirectoryI::addChild(const shared_ptr<NodePrx>& child)
```

```
{
    _contents.push_back(child);
}
```

See Also

- Slice for a Simple File System
- Example of a File System Client in C++11
- C++11 Mapping for Sequences
- slice2cpp Command-Line Options
- UUIDs as Identities in C++

Slice-to-C++11 Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to C++11 is identical to the mapping for these constructs without the `local` qualifier. The generated C++ code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

The rest of this section describes the mapping of the remaining local types to C++11:

C++11 Mapping for Local Interfaces

On this page:

Mapped C++ Class

A Slice local interface is mapped to a C++ abstract base class with the same name, for example:

Slice

```
module Ice
{
    local interface Communicator
    {
        ...
    }
}
```

is mapped to the C++ class `Communicator`:

C++

```
namespace Ice
{
    class Communicator
    {
        ...
    };
}
```

The `delegate` metadata allows you to map a local interface with a single operation to a `std::function`. For example:

Slice

```
module Ice
{
    ["delegate"] local interface ValueFactory
    {
        Value create(string type);
    }
}
```

is mapped to a function in C++:

C++

```
namespace Ice
{
    using ValueFactory = std::function<std::shared_ptr<Ice::Value>(const
std::string&)>;
}
```

The signature of this function is described on C++11 mapping for operations on local types.

LocalObject in C++

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to nothing at all in C++11, as shown in the example above.

Mapping for Local Interface Inheritance in C++

Inheritance of local Slice interfaces is mapped to public virtual inheritance in C++. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

C++

```
namespace M
{
    class A { ... };
    class B : public virtual A { ... };
    class C : public virtual A { ... };
    class D : public virtual B, public virtual C { ... };
}
```

C++11 Mapping for Local Classes

On this page:

Mapped C++ Class

A local Slice class is mapped to a C++ class with the same name. For example:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ...
    }
}
```

is mapped to the C++ class `ConnectionInfo`:

C++

```
namespace Ice
{
    class ConnectionInfo
    {
        ...
    };
}
```

LocalObject in C++

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to nothing in C++11.

Mapping for Local Interface Inheritance in C++

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` for classes is mapped to simple inheritance in C++, while `implements` is mapped to public virtual inheritance. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

C++

```
namespace M
{
    class A { ... };
    class B { ... };

    class C : public virtual A, public virtual B { ... };
    class D : public C { ... };
}
```

C++11 Mapping for Local Exceptions

On this page:

Mapped C++ Class

A local Slice exception is mapped to a C++ class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the C++ class `InitializationException`:

C++

```
namespace Ice
{
    class InitializationException : public Ice::LocalException
    {
        ...
    };
}
```

Base Class for Local Exceptions in C++

All mapped C++ exception classes derive directly or indirectly from `Ice::LocalException`:

C++

```
namespace Ice
{
    class LocalException : public Exception
    {
    public:
        LocalException(const char*, int);
        LocalException(const LocalException&) = default;
        virtual ~LocalException();

        std::unique_ptr<LocalException> ice_clone() const;

        static const std::string& ice_staticId();
    };
}
```

All these member functions are described on the C++11 Mapping for Exceptions page.

Mapping for Local Exception Inheritance in C++

A local Slice exception can extend another Slice exception. In C++, this is mapped to simple public inheritance. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

C++

```
namespace M
{
    class ErrorBase : public Ice::LocalException { ... };
    class ResourceError : public ErrorBase { ... };
}
```

C++11 Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to a pure virtual member function with the same name. The mapping of operation parameters to C++ is identical to the Client-Side Mapping for these parameters:

- in parameters are passed by value (for simple types such as integers), or by const reference (for other types)
- out parameters are passed by reference
- return values are mapped to return values in C++

However, unlike the Client-Side mapping, there is no trailing `Ice::Context` parameter in the mapped member functions.

The type of a parameter can be a local interface or class. Such a parameter is passed as a `std::shared_ptr<`*`mapped C++ class`*`>` for regular parameters (const reference for in parameters, reference for out parameters); a "delegate" interface parameter, mapped to a `std::function` in C++, is passed by value if it's an in parameter, and by reference if it's an out parameter.

A `LocalObject` parameter is mapped to a parameter of type `std::shared_ptr<void>`. You can also create constructed types (such as local sequences and local dictionaries) with local types.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

C++

```
namespace M
{
    class L;
    using LSeq = std::vector<std::shared_ptr<L>>;
    class L
    {
    public:
        virtual ~L() = default;
        virtual std::string op(int n, const std::string&, const
std::shared_ptr<void>& any,
                               int& m, std::string& t, LSeq& newLSeq) =
0;
    };
}
```

C++11 Mapping for Data Members in Local Types

Data members on local Slice types (classes, exceptions and structs) are mapped to C++ just like the data members of the corresponding non local Slice construct.

A local Slice type can have a data member of type local interface or class, which is mapped to a C++ data member of type `std::shared_ptr<`*`mapped C++ class`*`>`. A `LocalObject` data member is mapped to a C++ data member of the same name with type `std::shared_ptr<void>`.

Customizing the C++11 Mapping

Ice for C++ allows you to map Slice strings, sequences and dictionaries to your own C++ types, through the `cpp:type` and `cpp:view-type` metadata directives.

When you map a Slice type to a C++ type from the standard library type, Ice is able to marshal and unmarshal this type without your help. For example, if you map a `sequence<string>` to a `std::list<std::wstring>`, Ice knows exactly what to do. However, if you select a class or template outside the standard C++ library, such as your own C++ template, you usually need to tell Ice how to marshal and unmarshal this type.

The following pages describe the set of classes and templates that Ice uses for marshaling and unmarshaling, and how to plug-in your own C++ types into this framework.

Topics

The C++11 Stream Helpers

Ice for C++ uses a set of C++ templates and specializations of these templates to marshal and unmarshal C++ types. A type that can be marshaled and/or unmarshaled using this framework is known as a *Streamable.*

On this page:

Marshaling and Unmarshaling C++ Objects

Ice for C++ marshals C++ objects into streams of bytes, and unmarshals these C++ objects from streams of bytes. Ice for C++ has an internal implementation for these streams, and a publicly available implementation, and both implementations have mostly the same API.

To marshal a C++ object into a stream, you (or the generated code) can simply call `write(obj)` on that stream. For basic types, such as `int` or `string`, there is a corresponding write function on the stream. For other types, `write` is an inline template function that delegates to a specialization of the `StreamHelper` template class:

C++

```
namespace Ice
{
    class OutputStream : ...
    {
    public:
        ...
        template<typename T> inline void write(const T& v)
        {
            StreamHelper<T, StreamableTraits<T>::helper>::write(this,
v);
        }
    };
}
```

Likewise, when unmarshaling a C++ object from a stream, you (or the generated code) can simply call `read(obj)` on that stream, and the reading of most types delegates to a specialization of this `StreamHelper` template class:

C++

```
namespace Ice
{
    class InputStream : ...
    {
    public:
       ...
       template<typename T> inline void read(T& v)
       {
            StreamHelper<T, StreamableTraits<T>::helper>::read(this, v);
       }
    };
}
```

Ice looks for the `StreamHelper` specializations in namespace Ice, so they must all be in namespace Ice.

Each `StreamHelper` specialization typically provides a static `write` function to write the C++ object into the steam, and a static `read` function to read the C++ object from the stream. The stream object in these `StreamHelper` static functions is represented by a template parameter type, which allows a `StreamHelper` specialization to work with any stream implementation:

C++

```
// Typical StreamHelper specialization
namespace Ice
{
    template<>
    struct StreamHelper<SomeType, StreamHelperCategoryXXX>
    {
        template<class S> static inline void
        write(S* stream, const SomeType& v)
        {
            ... marshal v...
        }
        template<class S> static inline void
        read(S* stream, SomeType& v)
        {
            ... unmarshal v ...
        }
    };
}
```

The base `StreamHelper` template class is not defined, since there is no default way to read a C++ object from a stream or write this object into a stream:

C++

```
namespace Ice
{
    template<typename T, StreamHelperCategory st> struct StreamHelper;
}
```

For most types, Ice for C++ provides or generates the corresponding `StreamHelper` specialization (or partial specialization), for example all Slice sequences mapped to a C++ vector, C++ list or similar type use this specialization:

C++

```
namespace Ice
{
    template<typename T>
    struct StreamHelper<T, StreamHelperCategorySequence>
    {
        template<class S> static inline void
        write(S* stream, const T& v)
        {
            stream->writeSize(static_cast<Int>(v.size()));
            for(const auto& p : v)
            {
                stream->write(p);
            }
        }
        template<class S> static inline void
        read(S* stream, T& v)
        {
            int sz =
stream->readAndCheckSeqSize(StreamableTraits<typename
T::value_type>::minWireSize);
            T(sz).swap(v);
            for(auto& p : v)
            {
                stream->read(p);
            }
        }
    };
}
```

StreamableTraits

The `StreamHelper` specializations use specializations of the `StremableTraits` template class to retrieve some characteristics (or traits) of the C++ type it operates on:

C++

```
namespace Ice
{
    typedef int StreamHelperCategory;
    const StreamHelperCategory StreamHelperCategoryUnknown = 0;
    const StreamHelperCategory StreamHelperCategoryBuiltin = 1;
    const StreamHelperCategory StreamHelperCategoryStruct = 2;
    const StreamHelperCategory StreamHelperCategoryStructClass = 3;
    const StreamHelperCategory StreamHelperCategoryEnum = 4;
    const StreamHelperCategory StreamHelperCategorySequence = 5;
    const StreamHelperCategory StreamHelperCategoryDictionary = 6;
    const StreamHelperCategory StreamHelperCategoryProxy = 7;
    const StreamHelperCategory StreamHelperCategoryClass = 8;
    const StreamHelperCategory StreamHelperCategoryUserException = 9;

    template<typename T, typename Enabler = void>
    struct StreamableTraits
    {
        static const StreamHelperCategory helper = ...;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
}
```

In particular, `StreamHelper` specializations are divided in various categories, to allow a single partial specialization of `StreamHelper` to handle several similar C++ types. For example, Ice for C++ provides a single `StreamHelper` partial specialization for all `enum` types, with category `StreamHelperCategoryEnum`:

C++

```
template<typename T>
struct StreamHelper<T, StreamHelperCategoryEnum>
{
    template<class S> static inline void
    write(S* stream, const T& v)
    {
        ...sanity check...
        stream->writeEnum(static_cast<Int>(v),
StreamableTraits<T>::maxValue);
    }
    template<class S> static inline void
    read(S* stream, T& v)
    {
        intvalue = stream->readEnum(StreamableTraits<T>::maxValue);
        ..sanity check...
        v = static_cast<T>(value);
    }
};
```

`StreamHelperCategory` is an integer (`int`). Values between 0 and 20 are reserved for Ice; you can use other values for your own `StreamHelper` specializations.

A `StreamableTraits` is always defined in namespace Ice and provides three `static const` data members:

- `helper`
  The category of `StreamHelper` associated with this C++ type (see above). The base `StreamableTraits` template class computes `helper` automatically for maps, lists, vectors and similar types.

- `minWireSize`
  The minimum size (in bytes) of the corresponding Slice type when marshaled using the Ice Encoding. The base `StreamableTraits` template class provides the default value, 1 byte. This value should be one for sequences and dictionaries, since an empty sequence or dictionary is marshaled as a single byte (size = 0).

- `fixedLength`
  `true` when the corresponding Slice type is marshaled on a fixed number of bytes, and `false` when it is marshaled on a variable number of bytes. The default, provided by the base `StreamableTraits` template class, is false.

Ice provides `StreamableTraits` specializations for all the C++ types it uses when mapping Slice types to C++. For example, Ice provides a `StreamTraits<long long int>` (for 64-bit signed integers) and a `StreamableTraits<::std::string>`:

C++

```
namespace Ice
{
    template<>
    struct StreamableTraits<long long int>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryBuiltin;
        static const int minWireSize = 8;
        static const bool fixedLength = true;
    };

    template<>
    struct StreamableTraits<::std::string>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryBuiltin;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
}
```

For each constructed type, such as Slice `structs`, the Slice to C++ translator generates the corresponding `StreamableTraits` specialization. For example:

C++

```
// Generated C++ code
namespace Ice
{
    template<>
    struct StreamableTraits<::Ice::Identity>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryStruct;
        static const int minWireSize = 2;
        static const bool fixedLength = false;
    };
}
```

Marshaling and Unmarshaling Optional Values

As described in Data Encoding for Optional Values, the encoding or wire representation of an optional data member or parameter consists of:

- An optional format (a 3-bits value) that depends on the associated Slice type. For example, the optional format for a `sequence<string>` is `FSize` (or `6`).
- The tag associated with this optional data member or parameter (this tag is a positive integer)

- The actual value of this optional data member or parameter

To marshal an optional value into a stream, you (or the generated code) call `write(tag, obj)` on that stream, where `tag` is the optional's tag (an int), and obj is the optional value. `write` is a template function that uses a specialization of the `StreamOptionalHelper` template class when this optional value is set:

C++11

```
namespace Ice
{
    class OutputStream : ...
    {
    public:
    ...
        template<typename T> inline void write(int tag, const
Ice::optional<T>& v)
        {
            if(v)
            {
                writeOptional(tag, StreamOptionalHelper<T,
StreamableTraits<T>::helper,

StreamableTraits<T>::fixedLength>::optionalFormat);
                StreamOptionalHelper<T, StreamableTraits<T>::helper,
StreamableTraits<T>::fixedLength>::write(this, *v);
            }
        }
    };
}
```

Likewise, when unmarshaling an optional value from a stream, you (or the generated code) call `read(tag, obj)` on that stream, which delegates to a specialization of the same `StreamOptionalHelper` template class when the stream holds a value for this optional data member or parameter:

C++11

```
namespace Ice
{
    class InputStream : ...
    {
    public:
    ...
        template<typename T> inline void read(int tag, Ice::optional<T>&
v)
        {
            if(readOptional(tag, StreamOptionalHelper<T,
StreamableTraits<T>::helper,

StreamableTraits<T>::fixedLength>::optionalFormat))
            {
                v.emplace();
                StreamOptionalHelper<T, StreamableTraits<T>::helper,
StreamableTraits<T>::fixedLength>::read(this, *v);
            }
            else
            {
                v = Ice::nullopt;
            }
        }
    };
}
```

Each `StreamOptionalHelper` specialization usually provides `static read` and `write` functions, just like `StreamHelper` specializations, and also an `optionalFormat` static data member that provides the optional format of the corresponding Slice type.

For many types, the `StreamOptionalHelper` specialization simply delegates to this type's `StreamHelper` specialization, and computes `optionalFormat` with the `GetOptionalFormat` template:

C++11

```
 // GetOptionalFormat is declared but never defined
template<StreamHelperCategory st, int minWireSize, bool fixedLength>
struct GetOptionalFormat;

// Base StreamOptionalHelper template class
namespace Ice
{
    template<typename T, StreamHelperCategory st, bool fixedLength>
    struct StreamOptionalHelper
    {
        using Traits = StreamableTraits<T>;

        static const OptionalFormat optionalFormat =
GetOptionalFormat<st, Traits::minWireSize, fixedLength>::value;

        template<class S> static inline void
        write(S* stream, const T& v)
        {
            stream->write(v);
        }
        template<class S> static inline void
        read(S* stream, T& v)
        {
            stream->read(v);
        }
    };
}
```

For example, the optional format for a long long int (64-bit signed integer, encoded on exactly 8 bytes) is provided by this specialization of `GetOptionalFormat`:

C++

```
namespace Ice
{
    template<>
    struct GetOptionalFormat<StreamHelperCategoryBuiltin, 8, true>
    {
        static const OptionalFormat value = OptionalFormatF8;
    };
}
```

See Also

- Data Encoding
- Dynamic Ice

The cpp:type and cpp:view-type Metadata Directives with C++11

Ice for C++ provides two metadata directives that allow you to map Slice types to arbitrary C++ types: `cpp:type` and `cpp:view-type`. The se metadata directives currently apply only to the following Slice types:

- `string`
- `sequence`
- `dictionary`

On this page:

Customizing the sequence mapping with cpp:type

The `cpp:type:`*`c++-type`* metadata directive allows you to map a given Slice type, data member or parameter to the C++ type of your choice.

For example, you can override the default mapping of a Slice sequence type:

Slice

```
[["cpp:include:list"]]

module Food
{
    enum Fruit { Apple, Pear, Orange };

    ["cpp:type:std::list<Food::Fruit>"]
    sequence<Fruit> FruitPlatter;
}
```

With this metadata directive, the Slice sequence now maps to a C++ `std::list` instead of the default `std::vector`:

C++

```
#include <list>

namespace Food
{

    using FruitPlatter = std::list<Food::Fruit>;

    // ...
}
```

The Slice to C++ compiler takes the string following the `cpp:type:` prefix as the name of the mapped C++ type. For example, we could use `["cpp:type:::std::list< ::Food::Fruit>"]`. In that case, the compiler would use a fully-qualified name to define the type:

C++

```
using FruitPlatter = ::std::list< ::Food::Fruit>;
```

Note that the code generator inserts whatever string you specify following the `cpp:type:` prefix literally into the generated code. We recommend you use fully qualified names to avoid C++ compilation failures due to unknown symbols.

Also note that, to avoid compilation errors in the generated code, you must instruct the compiler to generate an appropriate include directive with the `cpp:include` global metadata directive. This causes the compiler to add the line

C++

```
#include <list>
```

to the generated header file.

In addition to modifying the type of a sequence itself, you can also modify the mapping for particular return values or parameters. For example:

Slice

```
[["cpp:include:list"]]
[["cpp:include:deque"]]

module Food
{
    enum Fruit { Apple, Pear, Orange }

    sequence<Fruit> FruitPlatter;

    interface Market
    {
        ["cpp:type:list<::Food::Fruit>"]
        FruitPlatter
barter(["cpp:type:deque<::Food::Fruit>"] FruitPlatter offer);
    }
}
```

With this definition, the default mapping of `FruitPlatter` to a C++ `vector` still applies but the return value of `barter` is mapped as a `list`, and the `offer` parameter is mapped as a `deque`.

Instead of `std::list` or `std::deque`, you can specify a type of your own as the sequence type, for example:

Slice

```
[["cpp:include:FruitBowl.h"]]

module Food
{
    enum Fruit { Apple, Pear, Orange }

    ["cpp:type:FruitBowl"]
    sequence<Fruit> FruitPlatter;
}
```

With these metadata directives, the compiler will use a C++ type `FruitBowl` as the sequence type, and add an `include` directive for the header file `FruitBowl.h` to the generated code.

The class or template class you provide must meet the following requirements:

- The class must have a default constructor.
- The class must have a copy constructor.

If you use a class that also meets the following requirements

- The class has a single-argument constructor that takes the size of the sequence as an argument of unsigned integral type.
- The class has a member function `size` that returns the number of elements in the sequence as an unsigned integral type.
- The class provides a member function `swap` that swaps the contents of the sequence with another sequence of the same type.
- The class defines `iterator` and `const_iterator` types and provides `begin` and `end` member functions with the usual semantics; its iterators are comparable for equality and inequality.

then you do not need to provide code to marshal and unmarshal your custom sequence – Ice will do it automatically.

Less formally, this means that if the provided class looks like a `vector`, `list`, or `deque` with respect to these points, you can use it as a custom sequence implementation without any additional coding.

Customizing the dictionary mapping with cpp:type

You can override the default mapping of Slice dictionaries to C++ maps with a `cpp:type` metadata directive, for example:

Slice

```
[["cpp:include:unordered_map"]]

["cpp:type:std::unordered_map<long long, Employee>"] dictionary<long,
Employee> EmployeeMap;
```

With this metadata directive, the dictionary now maps to a C++ `std::unordered_map`:

C++

```
#include <unordered_map>

using EmployeeMap = std::unordered_map<long long, Employee>;
```

Like with sequences, anything following the `cpp:type:` prefix is taken to be the name of the type. For example, we could use `["cpp:type`

`:::std::unordered_map<int, std::string>"]`. In that case, the compiler would use a fully-qualified name to define the type:

C++

```
using IntStringDict = ::std::unordered_map<int, std::string>;
```

To avoid compilation errors in the generated code, you must instruct the compiler to generate an appropriate include directive with the `cpp:include` global metadata directive. This causes the compiler to add the line

C++

```
#include <unordered_map>
```

to the generated header file.

Instead of `std::unordered_map`, you can specify a type of your own as the dictionary type, for example:

Slice

```
[["cpp:include:CustomMap.h"]]

["cpp:type:MyCustomMap<long long, Employee>"] dictionary<long, Employee>
EmployeeMap;
```

With these metadata directives, the compiler will use a C++ type `MyCustomMap` as the dictionary type, and add an include directive for the header file `CustomMap.h` to the generated code.

The class or template class you provide must meet the following requirements:

- The class must have a default constructor.
- The class must have a copy constructor.
- The class must provide nested types named `key_type`, `mapped_type` and `value_type`.
- The class must provide `iterator` and `const_iterator` types and provide `begin` and `end` member functions with the usual semantics; these iterators must be comparable for equality and inequality.
- The class must provide a `clear` function.
- The class must provide an `insert` function that takes an `iterator` (as location hint) plus a `value_type` parameter, and returns an `iterator` to the new entry or to the existing entry with the given key.

Less formally, this means you can use any class or template class that looks like a standard `map` or `unordered_map` as your custom dictionary type.

In addition to modifying the type of a dictionary itself, you can also modify the mapping for particular return values or parameters. For example:

Slice

```
[["cpp:include:unordered_map"]]

module HR
{
    struct Employee
    {
       long   number;
       string firstName;
       string lastName;
    }
    dictionary<long, Employee> EmployeeMap;

    interface Office
    {
        ["cpp:type:std::unordered_map<long long, Employee>"] EmployeeMap
getAllEmployees();
    }
}
```

With this definition, `getAllEmployees` returns an `unordered_map`, while other unqualified parameters of type `EmployeeMap` would use the default mapping (to a `std::map`).

cpp:type with custom C++ types

If your C++ type does not look like a standard C++ container, you need to tell Ice how to marshal and unmarshal this type by providing your own `StreamHelper` specialization for this type.

For example, you can map a Slice `sequence<byte>` to a Google Protocol Buffer C++ class, `tutorial:Person`, with the following metadata directive:

Slice

```
module Demo
{
   ["cpp:type:tutorial::Person"] sequence<byte> Person;
}
```

Since `tutorial::Person` does not look like a `vector<Ice::Byte>` or a `list<Ice::Byte>`, you need a `StreamHelper` specialization for `tutorial::Person`.

The simplest is to create a `StreamHelper` specialization that handles only this specific class:

C++

```
namespace Ice
{
    template<>
    struct StreamHelper<tutorial::Person, StreamHelperCategoryUnknown>
    {
        template<class S> static inline void
        write(S* stream, const tutorial::Person& v)
        {
            // ... marshal v into a sequence of bytes...
        }

        template<class S> static inline void
        read(S* stream, tutorial::Person& v)
        {
            //... unmarshal bytes from stream into v...
        }
    };
}
```

You should also provide the corresponding `StreamTraits` specialization:

C++

```
namespace Ice
{
    template<>
    struct StreamableTraits<tutorial::Person>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryUnknown;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
}
```

This `StreamTraits` specialization is actually optional for `tutorial::Person`, and more generally for any mapping of `sequence<byte>`, since these are the values provided by the base `StreamableTraits` template.

Finally, remember to insert the header for these `StreamHelper` and `StreamTraits` specializations with a `cpp:include` global metadata directive:

Slice

```
[["cpp:include:Person.pb.h"]]
[["cpp:include:PersonStreaming.h"]]
module Demo {
["cpp:type:tutorial::Person"] sequence<byte> Person;
```

Now, if your application maps several Slice `sequence<byte>` to different Google Protocol Buffers, you could provide a `StreamHelper` specialization for each of these Google Protocol Buffer classes, but it would be more judicious to provide a single partial specialization capable of marshaling and unmarshaling any Google Protocol Buffer C++ class as a Slice `sequence<byte>` in a stream:

C++

```
namespace Ice
{

    // A new helper category for all Google Protocol Buffers
    const StreamHelperCategory StreamHelperCategoryProtobuf = 100;

    // All classes derived from ::google::protobuf::MessageLite will use
this StreamableTraits
    template<typename T>
    struct StreamableTraits<T, typename std::enable_if<std::is_base_of<
::google::protobuf::MessageLite, T>::value >::type>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryProtobuf;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
    // T can be any Google Protocol Buffer C++ class
    template<typename T>
    struct StreamHelper<T, StreamHelperCategoryProtobuf>
    {
        template<class S> static inline void
        write(S* stream, const T& v)
        {
            std::vector<Byte> data(v.ByteSize());
            // ... marshal v into a sequence of bytes...
            stream->write(&data[0], &data[0] + data.size());
        }

        template<class S> static inline void
        read(S* stream, T& v)
        {
            std::pair<const Byte*, const Byte*> data;
            stream->read(data);
            //... unmarshal data into v...
        }
    };
}
```

If you use any of these `sequence<byte>` mapped to Google Protocol Buffers for optional data members or optional parameters, you also need to tell Ice which optional format to use:

C++

```
namespace Ice
{
    // Optional format for Slice sequence<byte> mapped to Google
Protocol Buffer
    // The template parameters correspond to the data members of the
corresponding StreamTraits specialization.
    template<>
    struct GetOptionalFormat<StreamHelperCategoryProtobuf, 1, false>
    {
        static const OptionalFormat value = OptionalFormatVSize;
    };
}
```

The `OptionalFormat` provided by `GetOptionalFormat` for `StreamHelperCategoryUnknown` and its default `StreamableTraits` is `OptionalFormatVSize`, like in the example above. This way, if your application needs a single `sequence<byte>` mapped to a custom C++ type, you can provide just a `StreamHelper` specialization for this type and rely on the default `StreamTraits` and `GetOptionalFormat` templates.

cpp:type:string and cpp:type:wstring

The metadata directives `cpp:type:string` and `cpp:type:wstring` are used to map Slice `strings` to `std::string` or `std::wstring`, as described in Alternative String Mapping for C++. The Slice to C++ compiler recognizes `string` and `wstring` as special tokens and does not treat them like C++ types with these names.

For example, you can map a `sequence<string>` to a `std::vector<std::wstring>` with either of the following directives:

Slice

```
// Special cpp:type:wstring metadata directive: it maps the sequence to
a std::vector<std::wstring>, not to a wstring!
["cpp:type:wstring"] sequence<string>;
```

or

Slice

```
// Maps the sequence to a std::vector<std::wstring> using a regular
cpp:type metadata directive
["cpp:type:std::vector<std::wstring>"] sequence<string>;
```

Avoiding copies with cpp:view-type

The main drawback of using standard C++ library classes to represent strings, sequences and dictionaries (as Ice does by default) is copying. For example, if we transfer an array of characters with Ice:

Slice

```
void sendChars(string s);
```

with the default mapping, our C++ code would look like:

C++

```
// client side
const char* cstring = ... null terminated array of chars;
proxy->sendChars(string(cstring));

// server side
void sendChars(string s, ...)
{
   // use s;
}
```

Each invocation triggers a number of copies:

- (client) the creation of the string on the client side makes a copy of the array of characters
- (client) `sendChars` copies the characters of the string into the client-side marshaling buffer
- (server) the unmarshaling code creates a string from the bytes in the server-side marshaling buffer

If instead of `std::string`, Ice could use a string type with view semantics–its instances point to memory owned by some other objects–we could avoid these copies. This is exactly what the `cpp:view-type` metadata directive allows us to do: select a custom mapping for Slice `strings`, `sequences` and `dictionaries`.

For example, we can map our Slice string parameter in the example above to a C++17 `std::string_view` (a class with view semantics):

Slice

```
void sendChars(["cpp:view-type:std::string_view"] string s);
```

Our C++ code is pretty much the same, but we avoid several copies:

C++

```
// client side
const char* cstring = ... null terminated array of chars;
proxy->sendChars(string_view(cstring)); // string_view points to the
characters cstring, without copy

// server side
void sendChars(string_view s, ...) // string_view points to bytes in the
server-side unmarshaling buffer
{
   // use s;
}
```

With this metadata directive, the only remaining copy during each invocation is:

- (client) `sendChars` copies the characters into the client-side marshaling buffer

This `cpp:view-type` metadata is very much like the `cpp:type` metadata directive described earlier, and just like for `cpp:type`, the Slice to C++ compiler uses the string provided after the `cpp:view-type:` prefix literally in the generated code. The compiler does not (and cannot) check that this string represents a valid C++ type, or a C++ type with view semantics.

There are however two significant differences between `cpp:view-type` and `cpp:type:`

- `cpp:view-type` can be applied only to operation parameters, while `cpp:type` can be applied to the definition of sequence and dictionary types, to operation parameters and to data members
- `cpp:view-type` changes the mapping of a parameter to the specified C++ type only when it is safe to use a view object (an object that does not own memory), while `cpp:type` changes the mapping all the time.

For example, if instead of sending an array of characters from a client to a server, we return an array of characters from the server to the client, without using AMI or AMD (the default):

Slice

```
string getChars();
```

With the default mapping, we get a `std::string` back, and each invocation makes a few copies:

C++

```
// client-side
string s = prx->getChars();

// server side
string
getChars(...)
{
   const char* cstring = ... null terminated array of chars;
   return cstring; // the conversion to string makes a copy
}
```

Now, if we map the returned string to a `string_view` with the `cpp:type` metadata directive:

Slice

```
["cpp:type:std::string_view"] string getChars(); // don't do this
```

we avoid some copies but our `string_view` now points to deallocated memory and our program will crash!

C++

```
// client-side
string_view s = prx->getChars(); // string_view points to the
client-side marshaling buffer, which is deallocated as soon as
getChars() returns

// server side
string_view
getChars(...)
{
   const char* cstring = ... null terminated array of chars;
   return cstring; // string_view points to (or may point to)
stack-allocated memory, reclaimed when getChars() returns
}
```

Never use the `cpp:type` metadata directive with a view type (a type that does not manage its memory); you should only use `cpp:view-type` with view types.

With `cpp:view-type`, the compiler changes the mapping only when it's safe to use a view-type, namely:

- Input parameters, on the client-side and on the server-side
- Out and return parameters provided by the Ice run-time to AMI callbacks
- Out and return parameters provided to marshaled results or AMD callbacks

The `cpp:array` metadata directive for sequences follows the same rules.

With our `getChars` example:

Slice

```
["cpp:view-type:std::string_view"] string getChars();
```

it is not safe the change the client-side mapping or the server-side mapping (unless we use a marshaled result or AMD), so the returned C++ type remains a `std::string`.

Like with the `cpp:type` metadata directive, if the C++ type specified with `cpp:view-type` is a standard library type (such as `std::list`) or looks like one, Ice will automatically marshal and unmarshal it for you. You just need to include the corresponding header with the `cpp:include` global metadata directive:

Slice

```
[["cpp:include:MyContainer.h"]]
module Sample
{
    ...
}
```

For other C++ types, you need to tell Ice how to marshal and unmarshal these types, as described above in cpp:type with custom C++ types.

In particular, Ice does not know how to marshal and unmarshal the `string_view` type. If you want to map some string parameters to `stri`

ng_view, you need to provide a StreamHelper for string_view, such as:

C++

```
namespace Ice
{
    template<>
    struct StreamableTraits<std::string_view>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryBuiltin;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };

    template<>
    struct StreamHelper<std::string_view, StreamHelperCategoryBuiltin>
    {
        template<class S> static inline void
        write(S* stream, const std::string_view& v)
        {
            stream->write(v.data(), v.size());
        }

        template<class S> static inline void
        read(S* stream, std::string_view& v)
        {
            const char* vdata = 0;
            size_t vsize = 0;
            stream->read(vdata, vsize);
            v = std::string_view(vadata, vsize);
        }
    };
}
```

This StreamHelper specialization will take care of plain string parameters, and also sequence or dictionary parameters that contain strings mapped to string_view (when it's safe to do so), such as:

Slice

```
["marshaled-result", "cpp:view-type:std::vector<std::string_view>"]
StringSeq
 echoStringSeq(["cpp:view-type:std::vector<std::string_view>"] StringSeq
seq);
```

The Ice/throughput demo illustrates the use of cpp:view-type with sequences of strings.

Using both cpp:type and cpp:view-type

If you specify both `cpp:type` and `cpp:view-type` for the same parameter, `cpp:view-type` applies to mapped parameters safe for view types, and `cpp:type` applies to all other mapped parameters.

With the following somewhat contrived example:

Slice

```
["marshaled-result", "cpp:view-type:std::vector<std::string_view>",
"cpp:type:std::list<std::wstring>"] StringSeq getStringSeq();
```

calling `getStringSeq()` on a proxy returns a `std::list<std::wstring>`, while the `StringSeq` is passed to the marshaled result struct as a `std::vector<string_view>`.

See Also

- Data Encoding
- Dynamic Ice

Version Information in C++11

Ice header files include the definitions of three macros that expand to the version of the Ice run time:

C++

```
#define ICE_STRING_VERSION "3.7.2" // "<major>.<minor>.<patch>"
#define ICE_INT_VERSION 30702      // AABBCC, with AA=major,
                                   // BB=minor, CC=patch

#define ICE_SO_VERSION "37"
```

`ICE_STRING_VERSION` is a string literal in the form *<major>.<minor>.<patch>*, for example, `3.7.2`. For alpha and beta releases, this string version is *<major>.<minor>[a|b]<number>*, for example, `3.7a3`.

`ICE_INT_VERSION` is an integer literal in the form *AABBCC*, where *AA* is the major version number, *BB* is the minor version number, and *CC* is the patch level, for example, `30602` for version 3.6.2. For alpha and beta releases, the patch level is set to 5x (alphas) or 6x (betas) so, for example, for version 3.7a3, the value is `30753`.

`ICE_SO_VERSION` is a string that contains the version in the soname for the Ice library. For releases, it's in the form "36" or "37", without the patch version number. For alpha and beta releases, it's identical to `ICE_STRING_VERSION` but without a dot.

slice2cpp Command-Line Options (C++11)

On this page:

`slice2cpp` Command-Line Options

The Slice-to-C++ compiler, `slice2cpp`, offers the following command-line options in addition to the standard options.

`--header-ext` *EXT*

Changes the file extension for the generated header files from the default h to the extension specified by *EXT*.

You can also change the header file extension with a global metadata directive:

Slice

```
[["cpp:header-ext:hpp"]]

// ...
```

Only one such directive can appear in each source file. If you specify a header extension on both the command line and with a metadata directive, the metadata directive takes precedence. This ensures that included Slice files that were compiled separately get the correct header extension (provided that the included Slice files contain a corresponding metadata directive). For example:

Slice

```
// File example.ice
#include <Ice/BuiltinSequences.ice>

// ...
```

Compiling this file with

```
$ slice2cpp --header-ext=hpp -I/opt/Ice/include example.ice
```

generates `example.hpp`, but the `#include` directive in that file is for `Ice/BuiltinSequences.h` (not `Ice/BuiltinSequences.hpp`) because `BuiltinSequences.ice` contains the metadata directive `[["cpp:header-ext:h"]]`.

You normally will not need to use this metadata directive. The directive is necessary only if:

- You `#include` a Slice file in one of your own Slice files.
- The included Slice file is part of a library you link against.

- The library ships with the included Slice file's header.
- The library header uses a different header extension than your own code.

For example, if the library uses `.hpp` as the header extension, but your own code uses `.h`, the library's Slice file should contain a `[["cpp:header?ext:hpp"]]` directive. (If the directive is missing, you can add it to the library's Slice file.)

`--source-ext` *`EXT`*

Changes the file extension for the generated source files from the default `cpp` to the extension specified by *`EXT`*.

`--add-header` *`HDR`*`[,`*`GUARD`*`]`

This option adds an include directive for the specified header at the beginning of the generated source file (preceding any other include directives). If *`GUARD`* is specified, the include directive is protected by the specified guard. For example, `--add-header precompiled.h,__PRECOMPILED_H__` results in the following directives at the beginning of the generated source file:

C++

```
#ifndef __PRECOMPILED_H__
#define __PRECOMPILED_H__
#include <precompiled.h>
#endif
```

The option can be repeated to create include directives for several files.

As suggested by the preceding example, this option is useful mainly to integrate the generated code with a compiler's precompiled header mechanism.

`--include-dir` *`DIR`*

Modifies `#include` directives in source files to prepend the path name of each header file with the directory *`DIR`*.

`--impl-c++11`

Generate sample implementation files for the C++11 mapping. This option will not overwrite an existing file.

`--impl-c++98`

Generate sample implementation files for the C++98 mapping. This option will not overwrite an existing file.

`--checksum`

Generate checksums for Slice definitions.

Include Directives

The `#include` directives generated by the Slice-to-C++ compiler can be a source of confusion if the semantics governing their generation are not well-understood. The generation of `#include` directives is influenced by the command-line options `-I` and `--include-dir`; these options are discussed in more detail below. The `--output-dir` option directs the translator to place all generated files in a particular directory, but has no impact on the contents of the generated code.

Given that the `#include` directives in header files and source files are generated using different semantics, we describe them in separate sections.

Header Files

In most cases, the compiler generates the appropriate `#include` directives by default. As an example, suppose file `A.ice` includes `B.ice` using the following statement:

Slice

```
// A.ice
#include <B.ice>
```

Assuming both files are in the current working directory, we run the compiler as shown below:

```
$ slice2cpp -I. A.ice
```

The generated file `A.h` contains this `#include` directive:

C++

```
// A.h
#include <B.h>
```

If the proper include paths are specified to the C++ compiler, everything should compile correctly.

Similarly, consider the common case where `A.ice` includes `B.ice` from a subdirectory:

Slice

```
// A.ice
#include <inc/B.ice>
```

Assuming both files are in the `inc` subdirectory, we run the compiler as shown below:

```
$ slice2cpp -I. inc/A.ice
```

The default output of the compiler produces this `#include` directive in `A.h`:

C++

```
// A.h
#include <inc/B.h>
```

Again, it is the user's responsibility to ensure that the C++ compiler is configured to find `inc/B.h` during compilation.

Now let us consider a more complex example, in which we do not want the `#include` directive in the header file to match that of the Slice file. This can be necessary when the organizational structure of the Slice files does not match the application's C++ code. In such a case, the user may need to relocate the generated files from the directory in which they were created, and the `#include` directives must be aligned with the new structure.

For example, let us assume that `B.ice` is located in the subdirectory `slice/inc`:

Slice

```
// A.ice
#include <slice/inc/B.ice>
```

However, we do not want the `slice` subdirectory to appear in the `#include` directive generated in the header file, therefore we specify the additional compiler option `-Islice`:

```
$ slice2cpp -I. -Islice slice/inc/A.ice
```

The generated code demonstrates the impact of this extra option:

C++

```
// A.h
#include <inc/B.h>
```

As you can see, the `#include` directives generated in header files are affected by the include paths that you specify when running the compiler. Specifically, the include paths are used to abbreviate the path name in generated `#include` directives.

When translating an `#include` directive from a Slice file to a header file, the compiler compares each of the include paths against the path of the included file. If an include path matches the leading portion of the included file, the compiler removes that leading portion when generating the `#include` directive in the header file. If more than one include path matches, the compiler selects the one that results in the shortest path for the included file.

For example, suppose we had used the following options when compiling `A.ice`:

```
$ slice2cpp -I. -Islice -Islice/inc slice/inc/A.ice
```

In this case, the compiler compares all of the include paths against the included file `slice/inc/B.ice` and generates the following directive:

C++

```
// A.h
#include <B.h>
```

The option `-Islice/inc` produces the shortest result, therefore the default path for the included file (`slice/inc/B.h`) is replaced with `B.h`.

In general, the `-I` option plays two roles: it enables the preprocessor to locate included Slice files, and it provides you with a certain amount of control over the generated `#include` directives. In the last example above, the preprocessor locates `slice/inc/B.ice` using the include path specified by the `-I.` option. The remaining `-I` options do not help the preprocessor locate included files; they are simply hints to the compiler.

Finally, we recommend using caution when specifying include paths. If the preprocessor is able to locate an included file via multiple include paths, it always uses the first include path that successfully locates the file. If you intend to modify the generated `#include` directives by specifying extra `-I` options, you must ensure that your include path hints match the include path selected by the preprocessor to locate the included file. As a general rule, you should avoid specifying include paths that enable the preprocessor to locate a file in multiple ways.

Source Files

By default, the compiler generates `#include` directives in source files using only the base name of the included file. This behavior is usually appropriate when the source file and header file reside in the same directory.

For example, suppose `A.ice` includes `B.ice` from a subdirectory, as shown in the following snippet of `A.ice`:

Slice

```
// A.ice
#include <inc/B.ice>
```

We generate the source file using this command:

```
$ slice2cpp -I. inc/A.ice
```

Upon examination, we see that the source file contains the following `#include` directive:

C++

```
// A.cpp
#include <B.h>
```

However, suppose that we wish to enforce a particular standard for generated `#include` directives so that they are compatible with our C++ compiler's existing include path settings. In this case, we use the `--include-dir` option to modify the generated code. For example, consider the compiler command shown below:

```
$ slice2cpp --include-dir src -I. inc/A.ice
```

The source file now contains the following `#include` directive:

C++

```
// A.cpp
#include <src/B.h>
```

Any leading path in the included file is discarded as usual, and the value of the `--include-dir` option is prepended.

See Also

- Using the Slice Compilers
- Using Slice Checksums in C++11

C++11 Strings and Character Encoding

On the wire, Ice transmits all strings as Unicode strings in UTF-8 encoding. For languages other than C++, Ice uses strings in their language-native Unicode representation and converts automatically to and from UTF-8 for transmission, so applications can transparently use characters from non-English alphabets.

However, for C++, how strings are represented inside a process depends on the platform as well as the mapping that is chosen for a particular string: the default mapping to `std::string`, or the alternative mapping to `std::wstring`.

> This discussion is only relevant for C++. For scripting language mappings based on Ice for C++, it is possible to use Ice's default string converter plug-in and to install your own string converter plug-in.

We will explore how strings are encoded by the Ice for C++ run time, and how you can achieve automatic conversion of strings in their native representation to and from UTF-8. For an example of using string converters in C++, refer to the sample program provided in the `demo/Ice/converter` subdirectory of your Ice distribution.

By default, the Ice run time encodes strings as follows:

- Narrow strings (that is, strings mapped to `std::string`) are presented to the application in UTF-8 encoding and, similarly, the application is expected to provide narrow strings in UTF-8 encoding to the Ice run time for transmission.

  With this default behavior, the application code is responsible for converting between the native codeset for 8-bit characters and UTF-8. For example, if the native codeset is ISO Latin-1, the application is responsible for converting between UTF-8 and narrow (8-bit) characters in ISO Latin-1 encoding.

  Also note that the default behavior does not require the application to do anything if it only uses characters in the ASCII range. (This is because a string containing only characters in the (7-bit) ASCII range is also a valid UTF-8 string.)

- Wide strings (that is, strings mapped to `std::wstring`) are automatically encoded as Unicode by the Ice run time as appropriate for the platform. For example, for Windows, the Ice run time converts between UTF-8 and UTF-16 in little-endian representation whereas, for Linux, the Ice run time converts between UTF-8 and UTF-32 in the endian-ness appropriate for the host CPU.

  With this default behavior, wide strings are transparently converted between their on-the-wire representation and their native C++ representation as appropriate, so application code need not do anything special. (The exception is if an application uses a non-Unicode encoding, such as Shift-JIS, as its native `wstring` codeset.)

Topics

See Also

Installing String Converters with C++11

The default behavior of the run time can be changed by providing application-specific string converters. If you install such converters, all Slice strings will be passed to the appropriate converter when they are marshaled and unmarshaled. Therefore, the string converters allow you to convert all strings transparently into their native representation without having to insert explicit conversion calls whenever a string crosses a Slice interface boundary.

You can install string converters by calling `setProcessStringConverter` for the narrow string converter, and `setProcessWstringConverter` for the wide string converter. Any strings that use the default (`std::string`) mapping are passed through the specified narrow string converter and any strings that use the wide (`std::wstring`) mapping are passed through the specified wide string converter. Passing `null` to one of these set functions resets the corresponding narrow or wide string converter to its default.

You can retrieve the previously installed string converters (or default string converters) with `getProcessStringConverter` and `getProcessWstringConverter`. The default narrow string converter is null, meaning all `std::string`s use the UTF-8 encoding. The default wide-string converter converts from UTF-16 or UTF-32 (depending on the size and endianness of the native wchar_t) to UTF-8.

The string converters are defined as follows:

C++

```
namespace Ice
{
    class UTF8Buffer
    {
    public:
        virtual Byte* getMoreBytes(size_t howMany,
Byte* firstUnused) = 0;
        virtual ~UTF8Buffer() = default;
    };

    template<typename charT>
    class BasicStringConverter
    {
    public:
        virtual Byte*
toUTF8(const charT* sourceStart, const charT* sourceEnd,
UTF8Buffer&) const = 0;

        virtual void fromUTF8(const Byte* sourceStart,
                              const Byte* sourceEnd,
                              std::basic_string<charT>& target) const =
0;

        virtual ~BasicStringConverter() = default;
    };

    typedef BasicStringConverter<char> StringConverter;
    typedef BasicStringConverter<wchar_t> WstringConverter;

    std::shared_ptr<StringConverter> getProcessStringConverter();
    void setProcessStringConverter(const
std::shared_ptr<StringConverter>&);

    std::shared_ptr<WstringConverter> getProcessWstringConverter();
    void setProcessWstringConverter(const
std::shared_ptr<WstringConverter>&);
}
```

As you can see, both narrow and wide string converters are simply templates with either a narrow or a wide character (`char` or `wchar_t`) as the template parameter.

Each communicator caches the narrow string converter and wide string converter installed when this communicator is initialized.

You should always install your string converters before creating your communicator(s). When using a plugin to set your string converters, you need to set the string converters in the constructor of your plugin class (which is executed when the plugin is loaded) and not in the initialization function of the plugin class (which is executed after the communicator has read and cached the process-wide string converters).

See Also

- Communicator Initialization
- C++11 Mapping for Built-In Types

UTF-8 Conversion with C++11

On this page:

Converting to UTF-8

If you have installed a string converter, the Ice run time calls the converter's `toUTF8` function whenever it needs to convert a native string into UTF-8 representation for transmission. The `sourceStart` and `sourceEnd` pointers point at the first byte and one-beyond-the-last byte of the source string, respectively. The implementation of `toUTF8` must return a pointer to the first unused byte following the converted string.

Your implementation of `toUTF8` must allocate the returned string by calling the `getMoreBytes` member function of the `UTF8Buffer` class that is passed as the third argument. (`getMoreBytes` throws a `std::bad_alloc` if it cannot allocate enough memory.) The `firstUnused` parameter must point at the first unused byte of the allocated memory region. You can make several calls to `getMoreBytes` to incrementally allocate memory for the converted string. If you do, `getMoreBytes` may relocate the buffer in memory. (If it does, it copies the part of the string that was converted so far into the new memory region.) The function returns a pointer to the first unused byte of the (possibly relocated) memory.

Conversion can also fail because the encoding of the source string is internally incorrect. In that case, you should throw a `Ice::IllegalConversionException` exception from `toUTF8`, for example:

C++

```
throw Ice::IllegalConversionException(__FILE__, __LINE__, "bad encoding
because ...");
```

After it has marshaled the returned string into an internal marshaling buffer, the Ice run time deallocates the string.

Converting from UTF-8

During unmarshaling, the Ice run time calls the `fromUTF8` member function on the corresponding string converter. The function converts a UTF-8 byte sequence into its native form as a `std::string` or `std::wstring`. The string into which the function must place the converted characters is passed to `fromUTF8` as the `target` parameter.

See Also

- Installing String Converters with C++11

String Parameters in Local C++11 Calls

In C++, and indirectly in Python, Ruby, and PHP, all Ice local APIs are narrow-string based, meaning you could not for example recompile `Properties.ice` to get property names and values as wide strings.

Installing a narrow-string converter could cause trouble for these local calls if UTF-8 conversion occurs in the underlying implementation. For example, the `stringToIdentity` operation creates an intermediary UTF-8 string. If this string contains characters that are not in your native codeset (as determined by the narrow-string converter), the `stringToIdentity` call will fail.

Likewise, when Ice reads properties from a configuration file, it converts the input (UTF-8 characters) into native strings. This conversion can also fail if the native encoding cannot convert some characters.

Most strings in local calls are never problematic because Ice does not perform any conversion, for example:

- adapter names in `createObjectAdapter`
- property names and values in `Properties`
- `ObjectAdapter::createProxy`, where the identity conversion occurs only when the proxy is marshaled

Finally, consider the Slice type `Ice::Context`, which is mapped in C++ as a `map<string, string>`. The mapping for `Context` cannot be changed to `map<wstring, wstring>`, therefore you cannot send or receive any context entry that is not in your narrow-string native encoding when a narrow-string converter is installed.

See Also

- Object Identity
- Properties and Configuration
- Request Contexts

Built-in String Converters in C++11

Ice provides three string converters to cover common conversion requirements:

- The Unicode wstring converter
  This is a string converter that converts between Unicode wide strings and UTF-8 strings. Unless you install a different string converter, this is the default converter that is used for wide strings.
- The iconv string converter (Linux, macOS, Unix)
  The `iconv` string converter converts strings using the `iconv` conversion facility. It can be used to convert either wide or narrow strings.
- The Windows string converter (Windows only)
  This string converter converts between multi-byte and UTF-8 strings and uses `MultiByteToWideChar` and `WideCharToMultiByte` for its implementation.

These string converters can be created through factory functions in the `Ice` namespace:

C++

```
namespace Ice
{
    shared_ptr<WstringConverter> createUnicodeWstringConverter();

    template<typename charT>
    shared_ptr<BasicStringConverter<charT>>
createIconvStringConverter(const std::string& internalCode = "");

    shared_ptr<StringConverter> createWindowsStringConverter(unsigned
int codepage);
}
```

See Also

- The C++11 iconv String Converter

C++11 String Conversion Convenience Functions

Ice provides two helper functions that allow you to convert between narrow strings and UTF-8 encoded strings using a string converter:

C++

```
namespace Ice
{
    std::string nativeToUTF8(const std::string&, const
shared_ptr<StringConverter>&);
    std::string UTF8ToNative(const std::string&, const
shared_ptr<StringConverter>&);
}
```

No conversion is performed when the provided string converter is null.

Ice provides two additional helper functions that allow you to convert between wide strings and narrow strings using the provided string converters:

C++

```
namespace Ice
{
    std::string wstringToString(const std::wstring&, const
shared_ptr<StringConverter>& = nullptr, const
shared_ptr<WstringConverter>& = nullptr);
    std::wstring stringToWstring(const std::string&, const
shared_ptr<StringConverter>& = nullptr, const
shared_ptr<WstringConverter>& = nullptr);
}
```

When the narrow string converter given to `wstringToString` is null, the encoding of the returned narrow string is UTF-8. When the wide string converter given to `wstringToString` is null, the encoding of wide string parameter is UTF-16 or UTF-32, depending on the size of `wchar_t`.

Likewise for `stringToWstring`, when the wide string converter is null, the encoding of the returned wide string is UTF-16 or UTF-32 depending on the size of wchar_t. When the narrow string converter given to `stringToWstring` is null, the encoding of narrow string parameter is UTF-8.

See Also

- UTF-8 Conversion with C++11

The C++11 iconv String Converter

For Linux, macOS and Unix platforms, `Ice` provides a string converter implementation that uses the `iconv` conversion facility to convert between the native encoding and UTF-8.

To use this string converter, you specify whether the conversion you want is for narrow or wide characters via the template argument, and you specify the corresponding native encoding with the constructor argument. For example, to create a converter that converts between ISO Latin-1 and UTF-8, you can instantiate the converter as follows:

C++

```
auto
stringConverter = Ice::createIconvStringConverter<char>("ISO-8859-1");
```

Similarly, to convert between the internal wide character encoding and UTF-8, you can instantiate a converter as follows:

C++

```
auto
wstringConverter = Ice::createIconvStringConverter<wchar_t>("WCHAR_T");
```

The string you pass to the factory function must be one of the values returned by `iconv -l`, which lists all the available character encodings for your machine. Passing no parameter or an empty string is equivalent to passing nl_langinfo(CODESET).

See Also

- Installing String Converters with C++11

The C++11 Ice String Converter Plug-in

The Ice run time includes a plug-in that supports conversion between UTF-8 and native encodings on Unix and Windows platforms. You can use this plug-in to install converters for narrow and wide strings into the communicator of an existing program. This feature is primarily intended for use in scripting language extensions such as Ice for Python; if you need to use string converters in your C++ application, we recommend using the technique described in Installing String Converters with C++11 instead.

> Ice for Python uses the C++98 mapping, so in practice Ice for Python could only use the C++ 98 Ice String Converter Plug-in.

Note that an application must be designed to operate correctly in the presence of a string converter. A string converter assumes that it converts strings in the native encoding into the UTF-8 encoding, and vice versa. An application that performs its own conversions on strings that cross a Slice interface boundary can cause encoding errors when those strings are processed by a converter.

Configuring the Ice String Converter Plug-in

You can install the plug-in using a configuration property like the one shown below:

```
Ice.Plugin.IceStringConverter=Ice:createStringConverter
iconv=encoding[,encoding] windows=code-page
```

The first component of the property value represents the plug-in's entry point, which includes the abbreviated name of the shared library or DLL (`Ice`) and the name of a factory function (`createStringConverter`).

The plug-in accepts the following arguments:

- `iconv=encoding[,encoding]`
  This argument is optional on Unix platforms and ignored on Windows platforms. If specified, it defines the `iconv` names of the narrow string encoding and the optional wide-string encoding. If this argument is not specified, the plug-in installs a narrow string converter that uses the default locale-dependent encoding.
- `windows=code-page`
  This argument is required on Windows platforms and ignored on Unix platforms. The *`code-page`* value represents a code page number, such as `1252`.

The plug-in's argument semantics are designed so that the same configuration property can be used on both Windows and Unix platforms, as shown in the following example:

```
Ice.Plugin.IceStringConverter=Ice:createStringConverter
iconv=ISO8859-1 windows=1252
```

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.IceStringConverter.cpp=Ice:createStringConverter
iconv=ISO8859-1 windows=1252
```

If using static libraries, you must also call the `Ice::registerIceStringConverter` function to ensure the plugin is linked with your application.

See Also

- UTF-8 Conversion with C++11
- Installing String Converters with C++11
- Plug-in Configuration
- Ice.InitPlugins
- Ice.Plugin.*

- Ice.PluginLoadOrder

Custom String Converter Plug-ins with C++11

If the default string converter plug-in does not satisfy your requirements, you can install your own string converters with a plugin, for example:

C++

```
class MyStringConverterPlugin : public Ice::Plugin
{
public:

    MyStringConverterPlugin(const shared_ptr<Ice::StringConverter>&
stringConverter,

const shared_ptr<Ice::WstringConverter>& wstringConverter = nullptr)
    {
        setProcessStringConverter(stringConverter);
        setProcessWstringConverter(wstringConverter);
    }

    virtual void initialize() override {}
    virtual void destroy() override {}
};
```

Like in this example, you should install the string converters in your plugin's constructor. Do not install the string converters in `initialize`.

In order to create such a plug-in, you must do the following:

- Define and export a factory function that returns an instance of your plugin class
- Implement the string converter(s) that you will pass to your plugin's constructor, or use the ones included with Ice.
- Package your code into a shared library or DLL.

To install your plug-in, use a configuration property like the one shown below:

```
Ice.Plugin.MyConverterPlugin=myconverter:createConverter ...
```

The first component of the property value represents the plug-in's entry point, which includes the abbreviated name of the shared library or DLL (`myconverter`) and the name of a factory function (`createConverter`).

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.MyConverterPlugin.cpp=myconverter:createConverter ...
```

See Also

- The C++11 Ice String Converter Plug-in
- Installing String Converters with C++11
- Plug-in API
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

C++11 Helper Functions

The C++11 mapping includes helper functions to simplify the comparison of `shared_ptr` targets:

C++

```
namespace Ice
{
    template<typename T, typename U>
    bool targetEqualTo(const T& lhs, const U& rhs);

    template<typename T, typename U>
    bool targetLess(const T& lhs, const U& rhs);

    template<typename T, typename U>
    bool targetGreater(const T& lhs, const U& rhs);

    template<typename T, typename U>
    bool targetLessEqual(const T& lhs, const U& rhs);

    template<typename T, typename U>
    bool targetGreaterEqual(const T& lhs, const U& rhs);

    template<typename T, typename U>
    bool targetNotEqualTo(const T& lhs, const U& rhs);
}
```

These functions can be used to compare any target types, but the most common use case is comparing proxies:

C++

```
shared_ptr<Ice::ObjectPrx> p1 = ...;
shared_ptr<Ice::ObjectPrx> p2 = ...;
if(Ice::targetEqualTo(p1, p2))
{
    ...
}
```

The helper functions rely on the target's implementation of the standard comparison operators `operator==` and `operator<`.

C++98 Mapping

Topics

Selecting the C++98 Mapping

Ice provides two distinct C++ mappings:

- C++98
  This was the only C++ mapping provided by Ice until version 3.6. This mapping relies only on features present in the ISO/IEC 14882:1998 C++ standard, informally known as C++98. It includes its own helper classes for smart pointers, threads, mutexes and so on.

- C++11
  This is a new mapping that takes advantage of features in the ISO/IEC 14882:2011 C++ standard, and occasionally newer features. This mapping requires a recent C++ compiler in C++11 or C++14/17 mode.

This chapter describes these two mappings in detail. When the same rule applies to both mappings, this rule is simply described as part of the "C++ mapping".

slice2cpp, the Slice-to-C++ translator, always generates code for both mappings, and C++ headers files provided by Ice, such as `Ice/Ice.h` and `IceGrid/IceGrid.h`, can be used with either mapping.

> The C++98 mapping is provided for backwards compatibility, and will not be included in future releases. If you are starting a new project with Ice, please use the C++11 mapping. If you are upgrading an existing application, you should consider upgrading your code to use the C++11 mapping.

Selecting the C++98 Mapping

The default Ice for C++ mapping is currently C++98, and you do not need to do anything special to select this mapping.

> The Ice C++11 and Ice C++98 libraries are built from the same source code, in ice/cpp. The resulting C++ libraries are nevertheless language mapping specific: `libIce++11.so` is for the Ice C++11 mapping, while `libIce.so` is for the Ice C++98 mapping.

Initialization and CommunicatorHolder in C++98

On this page:

Initializing the Ice Run Time with `Ice::initialize`

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `Ice::Communicator` object.

A `Communicator` is a local C++ object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application or program.

You initialize the Ice run time by calling the C++ function `Ice::initialize`, for example:

C++

```
int
main(int argc, char* argv[])
{
    Ice::CommunicatorPtr communicator = Ice::initialize(argc, argv);
    // ...
}
```

`initialize` accepts a C++ reference to `argc` and an argument vector `argv`. The function scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

> `Ice::initialize` has additional overloads to permit other information to be passed to the Ice run time.

`initialize` is a low-level function, as you need to explicitly call `destroy` on the returned `Communicator` object when you're done with Ice, typically just before returning from `main`. The `destroy` member function is responsible for finalizing the Ice run time. In particular, in a server, `destroy` waits for any operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory.

The general shape of the `main` function of an Ice-based application is therefore as follows:

C++

```
#include <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    int status = 0;
    try
    {
        Ice::CommunicatorPtr communicator = Ice::initialize(argc, argv);

        try
        {
            ... application code ...

            communicator->destroy(); // destroy is noexcept
        }
        catch(const std::exception&)
        {
            ...
            // make sure communicator is destroyed if an exception is
thrown
            communicator->destroy();
            throw;
        }
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        status = 1;
    }
    return status;
}
```

This code is a little bit clunky, as we need to make sure the communicator gets destroyed in all paths, including when an exception is thrown. As a result, most of the time, you should not call `initialize` directly: you should use instead a helper class that calls `initialize` and ensures the resulting communicator gets eventually destroyed.

`Ice::CommunicatorHolder` RAII Helper Class

A `CommunicatorHolder` is a small RAII helper class that creates a `Communicator` in its constructor (by calling `initialize`) and destroys this communicator in its destructor.

With a `CommunicatorHolder`, our typical `main` function becomes much simpler:

C++

```
#include <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    int status = 0;
    try
    {
        Ice::CommunicatorHolder ich(argc, argv); // Calls
Ice::initialize

        ... application code ...

        // CommunicatorHolder's destructor calls destroy on the
communicator
        // whether or not an exception is thrown
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        status = 1;
    }
    return status;
}
```

`Ice::CommunicatorHolder` is defined as follows:

C++

```
namespace Ice
{
    class CommunicatorHolder
    {
    public:

        CommunicatorHolder();

        explicit CommunicatorHolder(int& arrgc, const char* argv[],
const InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
        ... more CommunicatorHolder ctor overloads ...

        CommunicatorHolder(const CommunicatorPtr&);
        CommunicatorHolder& operator=(const CommunicatorPtr&);

        ~CommunicatorHolder();

        operator bool() const;
        const CommunicatorPtr& communicator() const;
        const CommunicatorPtr& operator->() const;
        CommunicatorPtr release();
        ...
    };
}
```

Let's examine each of these functions:

- `CommunicatorHolder()`
  This default constructor creates an empty holder (holds no Communicator).
- `CommunicatorHolder(int& argc, const char* argv[], const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION)`
  and many more constructors

  Each of these constructor calls `initialize` with the provided parameters; the new `CommunicatorHolder` then holds the resulting `Communicator` in a private data member (not shown).
- `CommunicatorHolder(const CommunicatorPtr&)`
  This constructor adopts the given communicator: the `CommunicatorHolder` becomes responsible to call `destroy` on it.
- `CommunicatorHolder& operator=(const CommunicatorPtr&)`
  This assignment operator destroys the communicator held by this `CommunicatorHolder`, then adopts the provided communicator.
- `~CommunicatorHolder()`
  The destructor calls `destroy` on the communicator held by this `CommunicatorHolder`.
- `operator bool() const`
  Returns `true` when this `CommunicatorHolder` holds a `Communicator`, and `false` otherwise.
- `const CommunicatorPtr& communicator() const`

  This function gives read-only access to the communicator held by this `CommunicatorHolder`.
- `const CommunicatoPtr& operator->() const`

This arrow operator allows you to use a `CommunicatorHolder` just like a `Communicator` object. For example:

C++

```
Ice::CommunicatorHolder ich(argc, argv);
Ice::ObjectPrx base = ich->stringToProxy("SimplePrinter:default -p
10000");
```

is equivalent to:

C++

```
Ice::CommunicatorHolder ich(argc, argv);
Ice::ObjectPrx base =
ich->communicator()->stringToProxy("SimplePrinter:default -p
10000");
```

- `CommunicatorPtr release()`
  This function returns the communicator held by `CommunicatorHolder` to the caller, and the caller becomes responsible to destroy this communicator. `CommunicatorHolder` no longer holds a communicator after this call.

The default constructor of `CommunicatorHolder` does nothing:

C++

```
Ice::CommunicatorHolder ich; // does not create a Communicator,
ich.communicator() returns a null shared_ptr
```

If you want to create a `CommunicatorHolder` that holds a `Communicator` created by `initialize` with no args, you can write:

C++

```
Ice::CommunicatorHolder ich = Ice::initialize();
```

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-C++98 Mapping

The client-side Slice-to-C++ mapping defines how Slice data types are translated to C++ types, and how clients invoke operations, pass parameters, and handle errors. Much of the C++ mapping is intuitive. For example, Slice sequences map to STL vectors, so there is essentially nothing new you have to learn in order to use Slice sequences in C++.

The rules that make up the C++ mapping are simple and regular. In particular, the mapping is free from the potential pitfalls of memory management: all types are self-managed and automatically clean up when instances go out of scope. This means that you cannot accidentally introduce a memory leak by, for example, ignoring the return value of an operation invocation or forgetting to deallocate memory that was allocated by a called operation.

The C++ mapping is fully thread-safe. For example, the reference counting mechanism for classes is interlocked against parallel access, so reference counts cannot be corrupted if a class instance is shared among a number of threads. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the C++ mapping, you should need no more than the Slice definition of your application and knowledge of the C++ mapping rules. In particular, looking through the generated header files in order to discern how to use the C++ mapping is likely to be confusing because the header files are not necessarily meant for human consumption and, occasionally, contain various cryptic constructs to deal with operating system and compiler idiosyncrasies. Of course, occasionally, you may want to refer to a header file to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **The `Ice` Namespace**
> All of the APIs for the Ice run time are nested in the `Ice` namespace, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` namespace are generated from Slice definitions; other parts of the `Ice` namespace provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` namespace throughout the remainder of the manual.

Topics

C++98 Mapping for Identifiers

A Slice identifier maps to an identical C++ identifier. For example, the Slice identifier `Clock` becomes the C++ identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a C++ keyword, the corresponding C++ identifier is prefixed with `_cpp_`. For example, the Slice identifier `while` is mapped as `_cpp_while`.

A single Slice identifier often results in several C++ identifiers. For example, for a Slice interface named `Foo`, the generated C++ code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_cpp_while` and `while Prx` (*not* `_cpp_whilePrx`), that is, the prefix is applied only to those generated identifiers that actually require it.

> You should try to avoid such identifiers as much as possible.

See Also

- Lexical Rules
- C++98 Mapping for Modules
- C++98 Mapping for Built-In Types
- C++98 Mapping for Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Sequences
- C++98 Mapping for Dictionaries
- C++98 Mapping for Constants
- C++98 Mapping for Exceptions

C++98 Mapping for Modules

A Slice module maps to a C++ namespace. The mapping preserves the nesting of the Slice definitions. For example:

Slice

```
module M1
{
    module M2
    {
        // ...
    }
    // ...
}

// ...

module M1    // Reopen M1
{
    // ...
}
```

This definition maps to the corresponding C++ definition:

C++

```
namespace M1
{
    namespace M2
    {
        // ...
    }
    // ...
}

// ...

namespace M1    // Reopen M1
{
    // ...
}
```

If a Slice module is reopened, the corresponding C++ namespace is reopened as well.

See Also

- Modules
- C++98 Mapping for Identifiers
- C++98 Mapping for Built-In Types
- C++98 Mapping for Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Sequences

C++98 Mapping for Built-In Types

On this page:

Mapping of Slice Built-In Types to C++ Types

The Slice built-in types are mapped to C++ types as shown in this table:

Slice	C++
bool	`bool`
byte	`Ice::Byte`
short	`Ice::Short`
int	`Ice::Int`
long	`Ice::Long`
float	`Ice::Float`
double	`Ice::Double`
string	`std::string`

> `Ice::Byte` is a typedef for `unsigned char`. This guarantees that byte values are always in the range 0..255.

All the basic types are guaranteed to be distinct C++ types, that is, you can safely overload functions that differ in only the types listed in the table above.

Alternative String Mapping for C++

You can use a metadata directive, `"cpp:type:wstring"`, to map strings to C++ `std::wstring`. For containers (such as interfaces or structures), the metadata directive applies to all strings within the container. A corresponding metadata directive, `"cpp:type:string"`, can be used to selectively override the mapping defined by the enclosing container. For example:

Slice

```
["cpp:type:wstring"]
struct S1
{
    string x;                          // Maps to std::wstring
    ["cpp:type:wstring"] string y;     // Maps to std::wstring
    ["cpp:type:string"] string z;      // Maps to std::string
}

struct S2
{
    string x;                          // Maps to std::string
    ["cpp:type:string"] string y;      // Maps to std::string
    ["cpp:type:wstring"] string z;     // Maps to std::wstring
}
```

With these metadata directives, the strings are mapped as indicated by the comments. By default, narrow strings are encoded as UTF-8, and wide strings use a UTF encoding that is appropriate for the platform on which the application executes. You can override the encoding for narrow and wide strings by registering a string converter with the Ice run time.

String View Mapping in C++

You can use the metadata directive `cpp:view-type:`*`string-view-type`* to map some string parameters to a custom C++ "view-type" of your choice. This view-type can reference memory without owning it, like the experimental `string_view` type. For example:

Slice

```
void sendString(["cpp:view-type:std::experimental::string_view"] string
data);
```

maps to:

C++

```
// Proxy function for synchronous call: input parameter mapped to
string_view type.
void sendString(const std::experimental::string_view&);
```

See Customizing the C++98 Mapping for a detailed description of the `cpp:view-type` metadata directive.

See Also

- Basic Types
- Customizing the C++98 Mapping
- C++98 Mapping for Identifiers
- C++98 Mapping for Modules
- C++98 Mapping for Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Sequences
- C++98 Mapping for Dictionaries
- C++98 Mapping for Constants

C++98 Mapping for Enumerations

A Slice enumeration maps to the corresponding enumeration in C++. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

Not surprisingly, the generated C++ definition is identical:

C++

```
enum Fruit { Apple, Pear, Orange };
```

Suppose we modify the Slice definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The generated C++ definition now includes an explicit initializer for every enumerator:

C++

```
enum Fruit { Apple = 0, Pear = 3, Orange = 4 };
```

See Also

- Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Sequences
- C++98 Mapping for Dictionaries

C++98 Mapping for Structures

A Slice structure maps to a C++ structure by default. In addition, you can use a metadata directive to map structures to C++ classes.

On this page:

Default Mapping for Structures in C++

Slice structures map to C++ structures with the same name. For each Slice data member, the C++ structure contains a public data member. For example, here is our Employee structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The Slice-to-C++ compiler generates the following definition for this structure:

C++

```
struct Employee
{
    Ice::Long   number;
    std::string firstName;
    std::string lastName;
    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
    bool operator<=(const Employee&) const;
    bool operator>(const Employee&) const;
    bool operator>=(const Employee&) const;
};
```

For each data member in the Slice definition, the C++ structure contains a corresponding public data member of the same name. Constructors are intentionally omitted so that the C++ structure qualifies as a *plain old datatype* (POD).

The structure may also contain comparison operators to allow its use as the key type of Slice dictionaries, which are mapped to `std::map` in C++. These operators have the following behavior:

- `operator==`
  Two structures are equal if (recursively), all its members are equal.
- `operator!=`
  Two structures are not equal if (recursively), one or more of its members are not equal.
- `operator<`
  `operator<=`
  `operator>`

`operator>=`

The comparison operators treat the members of a structure as sort order criteria: the first member is considered the first criterion, the second member the second criterion, and so on. Assuming that we have two `Employee` structures, `s1` and `s2`, this means that the generated code uses the following algorithm to compare `s1` and `s2`:

C++

```
bool Employee::operator<(const Employee& rhs) const
{
    if(this == &rhs)   // Short-cut self-comparison
    {
        return false;
    }

    // Compare first members
    //
    if(number < rhs.number)
    {
        return true;
    }
    else if(rhs.number < number)
    {
        return false;
    }

    // First members are equal, compare second members
    //
    if(firstName < rhs.firstName)
    {
        return true;
    }
    else if(rhs.firstName < firstName)
    {
        return false;
    }

    // Second members are equal, compare third members
    //
    if(lastName < rhs.lastName)
    {
        return true;
    }
    else if(rhs.lastName < lastName)
    {
        return false;
    }

    // All members are equal, so return false
    return false;
}
```

The comparison operators are only generated for structures that qualify as legal dictionary keys. You can force the Slice compiler to generate the comparison operators for non-qualifying structures by using the cpp:comparable metadata:

Slice

```
["cpp:comparable"]
struct NoKey
{
    float f;
}
```

Note that the ordering behavior for non-qualifying structures may not be deterministic.

Copy construction and assignment always have deep-copy semantics. You can freely assign structures or structure members to each other without having to worry about memory management. The following code fragment illustrates both comparison and deep-copy semantics:

C++

```
Employee e1, e2;
e1.firstName = "Bjarne";
e1.lastName = "Stroustrup";
e2 = e1;                        // Deep copy
assert(e1 == e2);
e2.firstName = "Andrew";        // Deep copy
e2.lastName = "Koenig";         // Deep copy
assert(e2 < e1);
```

Because strings are mapped to `std::string`, there are no memory management issues in this code and structure assignment and copying work as expected. (The default member-wise copy constructor and assignment operator generated by the C++ compiler do the right thing.)

Class Mapping for Structures in C++

Occasionally, the mapping of Slice structures to C++ structures can be inefficient. For example, you may need to pass structures around in your application, but want to avoid having to make expensive copies of the structures. (This overhead becomes noticeable for structures with many complex data members, such as sequences or strings.) Of course, you could pass the structures by const reference, but that can create its own share of problems, such as tracking the life time of the structures to avoid ending up with dangling references.

For this reason, you can enable an alternate mapping that maps Slice structures to C++ classes. Classes (as opposed to structures) are reference-counted. Because the Ice C++ mapping provides smart pointers for classes, you can keep references to a class instance in many places in the code without having to worry about either expensive copying or life time issues.

The alternate mapping is enabled by a metadata directive, `["cpp:class"]`. Here is our Employee structure once again, but this time with the additional metadata directive:

Slice

```
["cpp:class"] struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

Here is the generated class:

C++

```
class Employee : public IceUtil::Shared
{
public:
    Employee() {}
    Employee(::Ice::Long,
             const ::std::string&,
             const ::std::string&);
    ::Ice::Long number;
    ::std::string firstName;
    ::std::string lastName;

    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
    bool operator<=(const Employee&) const;
    bool operator>(const Employee&) const;
    bool operator>=(const Employee&) const;
};
```

Note that the generated class, apart from a default constructor, has a constructor that accepts one argument for each member of the structure. This allows you to instantiate and initialize the class in a single statement (instead of having to first instantiate the class and then assign to its members).

As for the default structure mapping, the class contains one public data member for each data member of the corresponding Slice structure.

The comparison operators behave as for the default structure mapping.

You can learn how to instantiate classes, and how to access them via smart pointers, in the sections describing the mapping for Slice classes — the API described there applies equally to Slice structures that are mapped to classes.

Default Constructors for Structures in C++

Structures have an implicit default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built?in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value.

If you declare a default value for at least one member of a structure, or use the class mapping for the structure, the Slice compiler also generates a second constructor. This *one-shot* constructor has one parameter for each data member, allowing you to construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members).

See Also

- Structures
- C++98 Mapping for Enumerations
- C++98 Mapping for Sequences
- C++98 Mapping for Dictionaries

C++98 Mapping for Sequences

On this page:

Default Sequence Mapping in C++

Here is the definition of our `FruitPlatter` sequence once more:

Slice

```
sequence<Fruit> FruitPlatter;
```

The Slice compiler generates the following definition for the `FruitPlatter` sequence:

C++

```
typedef std::vector<Fruit> FruitPlatter;
```

As you can see, the sequence simply maps to a standard `std::vector`, so you can use the sequence like any other vector. For example:

C++

```
// Make a small platter with one Apple and one Orange
//
FruitPlatter p;
p.push_back(Apple);
p.push_back(Orange);
```

Custom Sequence Mapping in C++

You can override the default mapping of Slice sequences to C++ vectors with the cpp:type and cpp:view-type metadata directives.

For example:

Slice

```
[["cpp:include:list"]]

module Food
{
    enum Fruit { Apple, Pear, Orange };

    ["cpp:type:std::list< ::Food::Fruit>"]
    sequence<Fruit> FruitPlatter;
}
```

With this metadata directive, the sequence now maps to a C++ `std::list`:

C++

```
#include <list>

namespace Food
{
    typedef std::list<Food::Fruit> FruitPlatter;

    // ...
}
```

Custom Mapping for Sequence Parameters in C++

In addition to the default and custom mappings of sequence types as a whole, you can use metadata to customize the mapping of a single operation parameter of type sequence.

Ice provides two metadata directives for this purpose, `["cpp:array"]` and `["cpp:range"]`.

Array Mapping for Sequence Parameters in C++

The array mapping for sequence parameters applies only to:

- In parameters, on the client-side and on the server-side
- Out and return parameters provided by the Ice run-time to AMI type-safe callbacks and AMI lambdas
- Out and return parameters provided to AMD callbacks

For example:

Slice

```
interface File
{
    void write(["cpp:array"] Ice::ByteSeq contents);
}
```

The `cpp:array` metadata directive instructs the compiler to map the `contents` parameter to a pair of pointers. With this directive, the `wri`

`te` method on the proxy has the following signature:

C++

```
void write(const std::pair<const Ice::Byte*,
const Ice::Byte*>& contents);
```

To pass a byte sequence to the server, you pass a pair of pointers; the first pointer points at the beginning of the sequence, and the second pointer points one element past the end of the sequence.

Similarly, for the server side, the `write` method on the skeleton has the following signature:

C++

```
virtual void write(const ::std::pair<const ::Ice::Byte*,
const ::Ice::Byte*>&,
                   const ::Ice::Current& = ::Ice::Current()) = 0;
```

The passed pointers denote the beginning and end of the sequence as a range `[first, last)` (that is, they use the usual semantics for iterators).

The array mapping is useful to achieve zero-copy passing of sequences. The pointers point directly into the server-side transport buffer; this allows the server-side run time to avoid creating a `vector` to pass to the operation implementation, thereby avoiding both allocating memory for the sequence and copying its contents into that memory.

> You can use the array mapping for any sequence type. However, it provides a performance advantage only for byte sequences (on all platforms) and for sequences of integral or floating point types (on some platforms).

Range Mapping for Sequence Parameters in C++

The range mapping for sequences is similar to the array mapping and exists for the same purpose, namely, to enable zero-copy of sequence parameters:

Slice

```
interface File
{
    void write(["cpp:range"] Ice::ByteSeq contents);
}
```

The `cpp:range` metadata directive instructs the compiler to map the `contents` parameter to a pair of `const_iterator`. With this directive, the `write` method on the proxy has the following signature:

C++

```
void write(const std::pair<Ice::ByteSeq::const_iterator,
Ice::ByteSeq::const_iterator>& contents);
```

Similarly, for the server side, the `write` method on the skeleton has the following signature:

C++

```
virtual void write(
    const ::std::pair<::Ice::ByteSeq::const_iterator,
::Ice::ByteSeq::const_iterator>&,
    const ::Ice::Current& = ::Ice::Current()) = 0;
```

The passed iterators denote the beginning and end of the sequence as a range `[first, last)` (that is, they use the usual semantics for iterators).

The motivation for the range mapping is the same as for the array mapping: the passed iterators point directly into the server-side transport buffer and so avoid the need to create a temporary `vector` to pass to the operation.

> As for the array mapping, the range mapping can be used with any sequence type, but offers a performance advantage only for byte sequences (on all platforms) and for sequences of integral type (x86 platforms only).

You can optionally add a type name to the `cpp:range` metadata directive, for example:

Slice

```
interface File
{
    void write(["cpp:range:std::deque<Ice::Byte>"]
Ice::ByteSeq contents);
}
```

This instructs the compiler to generate a pair of `const_iterator` for the specified type:

C++

```
virtual void write(
    const ::std::pair<std::deque<Ice::Byte>::const_iterator,
                      std::deque<Ice::Byte>::const_iterator>&,
    const ::Ice::Current& = ::Ice::Current()) = 0;
```

This is useful if you want to combine the range mapping with a custom sequence type that behaves like an standard container.

See Also

- Sequences
- Customizing the C++98 Mapping
- C++98 Mapping for Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Dictionaries
- C++98 Mapping for Operations

C++98 Mapping for Dictionaries

On this page:

Default Dictionary Mapping in C++

Here is the definition of our `EmployeeMap` once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

The following code is generated for this definition:

C++

```
typedef std::map<Ice::Long, Employee> EmployeeMap;
```

Again, there are no surprises here: a Slice dictionary simply maps to a standard `std::map`. As a result, you can use the dictionary like any other `map`, for example:

C++

```
EmployeeMap em;
Employee e;

e.number = 42;
e.firstName = "Stan";
e.lastName = "Lippman";
em[e.number] = e;

e.number = 77;
e.firstName = "Herb";
e.lastName = "Sutter";
em[e.number] = e;
```

Custom Dictionary Mapping in C++

You can override the default mapping of Slice dictionaries to C++ maps with a `cpp:type` a or `cpp:view-type` metadata directive, for example:

Slice

```
[["cpp:include:unordered_map"]]

["cpp:type:std::unordered_map<Ice::Long, Employee>"] dictionary<long,
Employee> EmployeeMap;
```

With this metadata directive, the dictionary now maps to a C++ `std::unordered_map`:

C++

```
#include <unordered_map>

typedef std::unordered_map<Ice::Long, Employee> EmployeeMap;
```

See Customizing the C++98 Mapping for detailed information about the `cpp:type` and `cpp:view-type` metadata directives.

See Also

- Dictionaries
- Customizing the C++98 Mapping
- C++98 Mapping for Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Sequences

C++98 Mapping for Constants

Slice constant definitions map to corresponding C++ constant definitions. For example:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Orange, Pear }
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

C++

```
const bool          AppendByDefault = true;
const Ice::Byte     LowerNibble =     15;
const std::string   Advice =          "Don't Panic!";
const Ice::Short    TheAnswer =       42;
const Ice::Double   PI =              3.1416;

enum Fruit { Apple, Orange, Pear };
const Fruit         FavoriteFruit =   Pear;
```

All constants are initialized directly in the header file, so they are compile-time constants and can be used in contexts where a compile-time constant expression is required, such as to dimension an array or as the `case` label of a `switch` statement.

Non-ASCII characters and universal character names in string literals are mapped to octal escapes in narrow C++ strings, and to universal character names in wide strings (except for characters in the range `\u0000` to `\u009f` that are mapped to octal escapes). For example:

Slice

```
const string Egg = "œuf";
const ["cpp:type:wstring"] string LargeEgg = "gros œuf";

const string Heart = "c\u0153ur";
const ["cpp:type:wstring"] string BigHeart = "grand c\u0153ur";
const ["cpp:type:wstring"] string Banana = "\U0001F34C";

const string DoubleTilde = "~\u007e";
const ["cpp:type:wstring"] string WDoubleTilde = "~\u007e";
```

is mapped to:

C++

```
const std::string Egg = "\305\223uf";
const std::wstring LargeEgg = L"gros \u0153uf";

const std::string Heart = "c\305\223ur";
const std::wstring BigHeart = L"grand c\u0153ur";
const std::wstring Banana = L"\U0001F34C";

const std::string DoubleTilde = "~\176";
const std::string WDoubleTilde = L"~\176";
```

See Also

- Constants and Literals
- C++98 Mapping for Identifiers
- C++98 Mapping for Modules
- C++98 Mapping for Built-In Types
- C++98 Mapping for Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Sequences
- C++98 Mapping for Dictionaries
- C++98 Mapping for Exceptions

C++98 Mapping for Exceptions

On this page:

C++ Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

C++

```
class GenericError : public Ice::UserException
{
public:
    std::string reason;

    GenericError() {}
    explicit GenericError(const std::string&);

    virtual std::string         ice_id() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
    // Other member functions here...
};

class BadTimeVal : public GenericError
{
public:
    BadTimeVal() {}
    explicit BadTimeVal(const std::string&);

    virtual std::string         ice_id() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
    // Other member functions here...
};

class BadZoneName : public GenericError
{
public:
    BadZoneName() {}
    explicit BadZoneName(const std::string&);

    virtual std::string         ice_id() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
};
```

Each Slice exception is mapped to a C++ class with the same name. For each exception member, the corresponding class contains a public data member. (Since `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.) Optional data members are mapped to instances of the `IceUtil::Optional` template.

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception has three additional member functions:

- `ice_id`
  As the name suggests, this member function returns the type ID of the exception. For example, if you call the `ice_id` member function of a `BadZoneName` exception defined in module `M`, it returns the string `"::M::BadZoneName"`. The `ice_id` member function is useful if you catch exceptions generically and want to produce a more meaningful diagnostic, for example:

C++

```
try
{
    // ...
}
catch(const GenericError& e)
{
    cerr << "Caught an exception: " << e.ice_id() << endl;
}
```

If an exception is raised, this code prints the id of the actual exception (such as `::M::BadTimeVal` or `::M::BadZoneName`). For exception that are not defined in Slice, `ice_id` returns the full name of the C++ class.

- `ice_file`
  `ice_file` returns the file name provided to the second constructor of `Exception`.
- `ice_line`
  `ice_line` returns the line number provided to the second constructor of `Exception`.
- `ice_print`
  The default implementation of `ice_print` prints the file name and line number (when available), and the type ID of the exception.
- `ice_stackTrace`
  The `ice_stackTrace` function returns the full stack trace when the exception was constructed, or an empty string, depending on the value of the `Ice.PrintStackTraces` property.
- `ice_clone`
  This member function allows you to polymorphically clone an exception. For example:

C++

```
try
{
    // ...
}
catch(const Ice::UserException& e)
{
   Ice::UserException* copy = e.clone();
}
```

`ice_clone` is useful if you need to make a copy of an exception without knowing its precise run-time type. This allows you to remember the exception and throw it later by calling `ice_throw`.

- `ice_throw`
  `ice_throw` allows you to throw an exception without knowing its precise run-time type. It is implemented as:

C++

```
void
GenericError::ice_throw() const
{
    throw *this;
}
```

You can call `ice_throw` to throw an exception that you previously cloned with `ice_clone`.

Each exception has a default constructor. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value. Optional data members are unset unless they declare default values.

An exception also has a second constructor that accepts one argument for each exception member. This constructor allows you to instantiate and initialize an exception in a single statement, instead of having to first instantiate the exception and then assign to its members. For each optional data member, its corresponding constructor parameter uses the same mapping as for operation parameters, allowing you to pass its initial value or `IceUtil::None` to indicate an unset value.

For derived exceptions, the constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

Note that the generated exception classes contain other member functions that are not shown here. However, those member functions are internal to the C++ mapping and are not meant to be called by application code.

All user exceptions ultimately inherit from `Ice::UserException`. In turn, `Ice::UserException` inherits from `Ice::Exception` (which is an alias for `IceUtil::Exception`):

C++

```
namespace IceUtil
{
    class Exception : public std::exception
    {
    public:

        virtual std::string        ice_id() const;
        Exception*                 ice_clone() const;
        void                       ice_throw() const;
        virtual void               ice_print(std::ostream&) const;
        // ...
    };
    std::ostream& operator<<(std::ostream&, const Exception&);
}

namespace Ice
{
    typedef IceUtil::Exception Exception;

    class UserException: public Exception
    {
        // ...
    };
}
```

`Ice::Exception` forms the root of the exception inheritance tree. Apart from the usual `ice_id`, `ice_clone`, and `ice_throw` member functions, it contains the `ice_print` member functions. The default implementation of `ice_print` prints the file name and line number (when available) and the type ID of the exception. For example, calling `ice_print` on a `BadTimeVal` exception defined in module `M` prints:

```
::M::BadTimeVal
```

To make printing more convenient, `operator<<` is overloaded for `Ice::Exception`, so you can also write:

C++

```
try
{
    // ...
}
catch(const Ice::Exception& e)
{
    cerr << e << endl;
}
```

This produces the same output because `operator<<` calls `ice_print` internally. You can optionally provide your own `ice_print` implem entation using the `cpp::ice_print` metadata directive.

For Ice run time exceptions, `ice_print` also shows the file name and line number at which the exception was thrown.

C++ Default Constructors for Exceptions

Exceptions have a default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value.

Exceptions also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assign to its members). For derived exceptions, this constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

C++ Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice::LocalException` (which, in turn, derives from `Ice::Exception`). `Ice::LocalException` has the usual member functions: `ice_id` `ice_clone`, `ice_throw`, and (inherited from `Ice::Exception`), `ice_print`, `ice_file`, and `ice_line`.

Recall the inheritance diagram for user and run-time exceptions. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice::Exception`
  This is the root of the complete inheritance tree. Catching `Ice::Exception` catches both user and run-time exceptions. As shown earlier, `Ice::Exception` is a typedef for `IceUtil::Exception`, and `IceUtil::Exception` inherits from `std::exception.`
- `Ice::UserException`
  This is the root exception for all user exceptions. Catching `Ice::UserException` catches all user exceptions (but not run-time exceptions).
- `Ice::LocalException`
  This is the root exception for all run-time exceptions. Catching `Ice::LocalException` catches all run-time exceptions (but not user exceptions).
- `Ice::TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `Ice::ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalExceptio n`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- C++98 Mapping for Identifiers
- C++98 Mapping for Modules
- C++98 Mapping for Built-In Types
- C++98 Mapping for Enumerations
- C++98 Mapping for Structures
- C++98 Mapping for Sequences
- C++98 Mapping for Dictionaries
- C++98 Mapping for Constants
- C++98 Mapping for Optional Values
- Versioning
- Object Life Cycle

C++98 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Classes and Proxy Handles

On the client side, a Slice interface maps to a class with member functions that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
module M
{
    interface Simple
    {
        void op();
    }
}
```

The Slice compiler generates the following definitions for use by the client:

C++

```
namespace IceProxy
{
    namespace M
    {
        class Simple;
    }
}

namespace M
{
    class Simple;
    typedef IceInternal::ProxyHandle< ::IceProxy::M::Simple> SimplePrx;
    typedef IceInternal::Handle< ::M::Simple> SimplePtr;
}

namespace IceProxy
{
    namespace M
    {
        class Simple : public virtual IceProxy::Ice::Object
        {
        public:
            void op();
            void op(const Ice::Context&);
            // ...
        };
    };
}
```

As you can see, the compiler generates a *proxy class* `Simple` in the `IceProxy::M` namespace, as well as a *proxy* handle `M::SimplePrx`. In general, for a module `M`, the generated names are `::IceProxy::M::<`*`interface-name`*`>` and `::M::<`*`interface-name`*`>Prx`.

In the client's address space, an instance of `IceProxy::M::Simple` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Inheritance from `Ice::Object`

`Simple` inherits from `IceProxy::Ice::Object`, reflecting the fact that all Ice interfaces implicitly inherit from `Ice::Object`. For each operation in the interface, the proxy class has two overloaded member functions of the same name. For the preceding example, we find that the operation `op` has been mapped to two member functions `op`.

One of the overloaded member functions has a trailing parameter of type `Ice::Context`. This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist. (The parameter is also used by IceStorm.)

Interface Inheritance

Inheritance relationships among Slice interfaces are maintained in the generated C++ classes. For example:

Slice

```
module M
{
    interface A { ... }
    interface B { ... }
    interface C extends A, B { ... }
}
```

The generated code for `C` reflects the inheritance hierarchy:

C++

```
namespace IceProxy
{
    namespace M
    {
        class C : public virtual A, public virtual B
        {
            ...
        };
    }
}
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

Proxy Handles

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. The following code will not compile because the proxy class does not provide a public default constructor:

C++

```
IceProxy::M::Simple s;  // Compile-time error!
```

Proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. When the client receives a proxy from the run time, it is given a *proxy handle* to the proxy, of type `<`*`interface-name`*`>Prx` (`SimplePrx` for the preceding example). The client accesses the proxy via its proxy handle; the handle takes care of forwarding operation invocations to its underlying proxy, as well as reference-counting the proxy. This means that no memory-management issues can arise: deallocation of a proxy is automatic and happens once the last handle to the proxy disappears (goes out of scope).

Because the application code always uses proxy handles and never touches the proxy class directly, we usually use the term proxy to denote both proxy handle and proxy class. This reflects the fact that, in actual use, the proxy handle looks and feels like the underlying proxy class instance. If the distinction is important, we use the terms *proxy class*, *proxy class instance*, and *proxy handle*.

Methods on Proxy Handles

As we saw for the preceding example, the handle is actually a template of type `IceInternal::ProxyHandle` that takes the proxy class as the template parameter. This template has the usual default constructor, copy constructor, and assignment operator.

Default constructor

You can default-construct a proxy handle. The default constructor creates a proxy that points nowhere (that is, points at no object at all). If you invoke an operation on such a null proxy, you get an `IceUtil::NullHandleException`:

C++

```
try
{
    SimplePrx s;        // Default-constructed proxy
    s->op();            // Call via nil proxy
    assert(0);          // Can't get here
}
catch(const IceUtil::NullHandleException&)
{
    cout << "As expected, got a NullHandleException" << endl;
}
```

Copy constructor

The copy constructor ensures that you can construct a proxy handle from another proxy handle. Internally, this increments a reference count on the proxy; the destructor decrements the reference count again and, once the count drops to zero, deallocates the underlying proxy class instance. That way, memory leaks are avoided:

C++

```
{                               // Enter new scope
    SimplePrx s1 = ...;         // Get a proxy from somewhere
    SimplePrx s2(s1);           // Copy-construct s2
    assert(s1 == s2);           // Assertion passes
}                               // Leave scope; s1, s2, and the
                                // underlying proxy instance
                                // are deallocated
```

Note the assertion in this example: proxy handles support comparison.

Assignment operator

You can freely assign proxy handles to each other. The handle implementation ensures that the appropriate memory-management activities take place. Self-assignment is safe and you do not have to guard against it:

C++

```
SimplePrx s1 = ...;     // Get a proxy from somewhere
SimplePrx s2;           // s2 is nil
s2 = s1;                // both point at the same object
s1 = 0;                 // s1 is nil
s2 = 0;                 // s2 is nil
```

Widening assignments work implicitly. For example, if we have two interfaces, `Base` and `Derived`, we can widen a `DerivedPrx` to a `BasePrx` implicitly:

C++

```
BasePrx base;
DerivedPrx derived;
base = derived;         // Fine, no problem
derived = base;         // Compile-time error
```

Implicit narrowing conversions result in a compile error, so the usual C++ semantics are preserved: you can always assign a derived type to a base type, but not vice versa.

Checked cast

Proxy handles provide a `checkedCast` method:

C++

```
namespace IceInternal
{
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T>
    {
    public:
        template<class Y>
        static ProxyHandle checkedCast(const ProxyHandle<Y>& r);

        template<class Y>

static ProxyHandle checkedCast(const ProxyHandle<Y>& r, const ::Ice::Co
ntext& c);

        // ...
    };
}
```

A checked cast has the same function for proxies as a C++ `dynamic_cast` has for pointers: it allows you to assign a base proxy to a derived proxy. If the type of the base proxy's target object is compatible with the derived proxy's static type, the assignment succeeds and, after the assignment, the derived proxy denotes the same object as the base proxy. Otherwise, if the type of the base proxy's target object is incompatible with the derived proxy's static type, the derived proxy is set to null. Here is an example to illustrate this:

C++

```
BasePrx base = ...;     // Initialize base proxy
DerivedPrx derived = DerivedPrx::checkedCast(base);
if(derived)
{
    // Base has run-time type Derived,
    // use derived...
}
else
{
    // Base has some other, unrelated type
}
```

The expression `DerivedPrx::checkedCast(base)` tests whether `base` points at an object of type `Derived` (or an object with a type that is derived from `Derived`). If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to the null proxy.

Note that `checkedCast` is a static member function so, to do a down-cast, you always use the syntax <*interface-name*>`Prx::checkedCast`.

Also note that you can use proxies in boolean contexts. For example, `if (proxy)` returns true if the proxy is not null.

A `checkedCast` typically results in a remote message to the server.The message effectively asks the server "is the object denoted by this reference of type Derived?"

> Calling `checkedCast` on a proxy that is already of the desired proxy type returns immediately that proxy. Otherwise, `checkedCast` always calls `ice_isA` on the target object, and upon success, creates a new instance of the desired proxy class.

The reply from the server is communicated to the application code in form of a successful (non-null) or unsuccessful (null) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of the target object is without confirmation from the server. (For example, the server may replace the implementation of the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive a `ConnectFailedException;` if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ObjectNotExistException`.

Unchecked cast

In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

C++

```
namespace IceInternal
{
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T>
    {
    public:
        template<class Y>
        static ProxyHandle uncheckedCast(const ProxyHandle<Y>& r);
        // ...
    };
}
```

An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object, for example:

C++

```
BasePrx base = ...;     // Initialize to point at a Derived
DerivedPrx derived;
derived = DerivedPrx::uncheckedCast(base);
// Use derived...
```

You should use an `uncheckedCast` only if you are certain that the target object indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not checked in any way; it does not contact the object in the server and, if it fails, it does not return null. If you use the proxy resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `OperationNotExistException`, but, depending on the circumstances, the Ice run time may also report an exception indicating that unmarshaling has failed, or even silently return garbage results.

> Calling `uncheckedCast` on a proxy that is already of the desired proxy type returns immediately that proxy. Otherwise, `uncheckedCast` creates a new instance of the desired proxy class.

Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization, it is common to receive a proxy of static type `Ice::Object`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

Stream insertion and stringification

For convenience, proxy handles also support insertion of a proxy into a stream, for example:

C++

```
Ice::ObjectPrx p = ...;
cout << p << endl;
```

This code is equivalent to writing:

C++

```
Ice::ObjectPrx p = ...;
cout << p->ice_toString() << endl;
```

Either code prints the stringified proxy. You could also achieve the same thing by writing:

C++

```
Ice::ObjectPrx p = ...;
cout << communicator->proxyToString(p) << endl;
```

The advantage of using the `ice_toString` member function instead of `proxyToString` is that you do not need to have the communicator available at the point of call.

Static id

The `ice_staticId` method returns the type ID of the proxy's Slice interface:

C++

```
namespace IceInternal
{
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T>
    {
    public:
        static const std::string& ice_staticId();
        // ...
    };
}
```

Here's an example that shows how to use it:

C++

```
BasePrx base = ...;
if(base->ice_id() == Derived::ice_staticId())
    // target object implements Derived
```

Applications normally use `checkedCast` instead of calling `ice_id` directly.

Using Proxy Methods in C++

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

C++

```
Ice::ObjectPrx proxy = communicator->stringToProxy(...);
proxy = proxy->ice_timeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

C++

```
Ice::ObjectPrx base = communicator->stringToProxy(...);
HelloPrx hello = HelloPrx::checkedCast(base);
hello = hello->ice_timeout(10000); // Type is preserved
hello->sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in C++

Proxy handles support comparison using the following operators:

- `operator bool`
  Proxies have a conversion operator to `bool`. The operator returns true if a proxy is not null, and false otherwise. This allows you to write:

  C++

  ```
  BasePrx base = ...;
  if(base)
          // It's a non-nil proxy
  if(!base)
          // It's a nil proxy
  ```

- `operator==`
  `operator!=`
  These operators permit you to compare proxies for equality and inequality. It's also legal to compare a proxy against the literal `0` to test whether a proxy is null, but we recommend using the `bool` operator instead.

- `operator<`
  `operator<=`
  `operator>`
  `operator>=`
  Proxies support comparison. This allows you to place proxies into STL containers such as maps or sorted lists.

Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with == and `!=` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

C++

```
Ice::ObjectPrx p1 = ...;        // Get a proxy...
Ice::ObjectPrx p2 = ...;        // Get another proxy...

if(p1 != p2)
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with ==, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with ==, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` namespace:

C++

```
namespace Ice
{
    bool proxyIdentityLess(const ObjectPrx&, const ObjectPrx&);
    bool proxyIdentityEqual(const ObjectPrx&, const ObjectPrx&);
    bool proxyIdentityAndFacetLess(const ObjectPrx&, const ObjectPrx&);
    bool proxyIdentityAndFacetEqual(const ObjectPrx&, const ObjectPrx&);
}
```

The `proxyIdentityEqual` function returns true if the object identities embedded in two proxies are the same and ignores other information in the proxies, such as facet and transport information. To include the facet name in the comparison, use `proxyIdentityAndFacetEqual` instead.

The `proxyIdentityLess` function establishes a total ordering on proxies. It is provided mainly so you can use object identity comparison with STL sorted containers. (The function uses `name` as the major ordering criterion, and `category` as the minor ordering criterion.) The `proxyIdentityAndFacetLess` function behaves similarly to `proxyIdentityLess`, except that it also compares the facet names of the proxies when their identities are equal.

`proxyIdentityEqual` and `proxyIdentityAndFacetLess` allow you to correctly compare proxies for object identity. The example below demonstrates how to use `proxyIdentityEqual`:

C++

```
Ice::ObjectPrx p1 = ...;        // Get a proxy...
Ice::ObjectPrx p2 = ...;        // Get another proxy...

if(!Ice::proxyIdentityEqual(p1, p2)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- C++98 Mapping for Operations
- Example of a File System Client in C++98
- Versioning
- IceStorm

C++98 Mapping for Operations

On this page:

Basic C++ Mapping for Operations

As we saw in the C++ mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy handle. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The proxy class for the `Node` interface, tidied up to remove irrelevant detail, is as follows:

C++

```
namespace IceProxy
{
    namespace Filesystem
    {
        class Node : virtual public IceProxy::Ice::Object
        {
        public:
            std::string name();
            // ...
        };
        typedef IceInternal::ProxyHandle<Node> NodePrx;
        // ...
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

C++

```
NodePrx node = ...;             // Initialize proxy
string name = node->name();     // Get name via RPC
```

The proxy handle overloads `operator->` to forward method calls to the underlying proxy class instance which, in turn, sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

Because the return value is of type `string`, it is safe to ignore the return value. For example, the following code contains no memory leak:

C++

```
NodePrx node = ...;             // Initialize proxy
node->name();                   // Useless, but no leak
```

This is true for all mapped Slice types: you can safely ignore the return value of an operation, no matter what its type — return values are always returned by value. If you ignore the return value, no memory leak occurs because the destructor of the returned value takes care of deallocating memory as needed.

Normal and `idempotent` Operations in C++

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy methods is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent string op2();
    idempotent void op3(string s);
}
```

The proxy class for this interface looks like this:

C++

```
namespace IceProxy
{
    class Example : virtual public IceProxy::Ice::Object
    {
    public:
        std::string op1();
        std::string op2();              // idempotent
        void op3(const std::string&);   // idempotent
        // ...
    };
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the `idempotent` keyword.

Passing Parameters in C++

In-Parameters in C++

The parameter passing rules for the C++ mapping are very simple: parameters are passed either by value (for small values) or by `const` reference (for values that are larger than a machine word). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following code for this definition:

C++

```
struct NumberAndString
{
    Ice::Int x;
    std::string str;
    // ...
};

typedef std::vector<std::string> StringSeq;

typedef std::map<Ice::Long, StringSeq> StringTable;

namespace IceProxy
{
    class ClientToServer : virtual public IceProxy::Ice::Object
    {
    public:
        void op1(Ice::Int, Ice::Float, bool, const std::string&);
        void op2(const NumberAndString&, const StringSeq&,
const StringTable&);
        void op3(const ClientToServerPrx&);
        // ...
    };
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

C++

```
ClientToServerPrx p = ...;              // Get proxy...

p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14;
bool b = true;
string s = "Hello world!";
p->op1(i, f, b, s);                     // Pass simple variables

NumberAndString ns = { 42, "The Answer" };
StringSeq ss;
ss.push_back("Hello world!");
StringTable st;
st[0] = ss;
p->op2(ns, ss, st);                     // Pass complex variables

p->op3(p);                              // Pass proxy
```

You can pass either literals or variables to the various operations. Because everything is passed by value or `const` reference, there are no memory-management issues to consider.

Out-Parameters in C++

The C++ mapping passes out-parameters by reference. Here is the Slice definition once more, modified to pass all parameters in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns, out StringSeq ss,
out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Slice compiler generates the following code for this definition:

C++

```
namespace IceProxy
{
    class ServerToClient : virtual public IceProxy::Ice::Object
    {
    public:
       void op1(Ice::Int&, Ice::Float&, bool&, std::string&);
       void op2(NumberAndString&, StringSeq&, StringTable&);
       void op3(ServerToClientPrx&);
       // ...
    };
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

C++

```
ServerToClientPrx p = ...;      // Get proxy...

int i;
float f;
bool b;
string s;

p->op1(i, f, b, s);
// i, f, b, and s contain updated values now

NumberAndString ns;
StringSeq ss;
StringTable st;

p->op2(ns, ss, st);
// ns, ss, and st contain updated values now

p->op3(p);
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes variables to an operation; once the operation completes, the values of those variables will be set by the server.

It is worth having another look at the final call:

C++

```
p->op3(p);      // Weird, but well-defined
```

Here, p is the proxy that is used to dispatch the call. That same variable p is also passed as an out-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe: the Ice run time will correctly handle all locking and memory-management activities.

Optional Parameters in C++

The mapping for optional parameters is the same as for required parameters, except each optional parameter is encapsulated in an `IceUtil::Optional` value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

The C++ mapping for this operation is shown below:

C++

```
IceUtil::Optional<Ice::Int>
execute(const IceUtil::Optional<std::string>& params,
IceUtil::Optional<Ice::Float>& value, ...);
```

The constructors provided by the `IceUtil::Optional` template simplify the use of optional parameters:

C++

```
IceUtil::Optional<Ice::Int> i;
IceUtil::Optional<Ice::Float> v;

i = proxy->execute("--file log.txt", v); // string converted to
Optional<string>
i = proxy->execute(IceUtil::None, v);    // params is unset

if(v)
    cout << "value = " << v.get() << endl;
```

For an optional output parameter, the Ice run time resets the client's `Optional` instance if the server does not supply a value for the parameter, therefore it is safe to pass an `Optional` instance that already has a value.

A well-behaved program must not assume that an optional parameter always has a value.

Chained Invocations in C++

Consider the following simple interface containing two operations, one to set a value and one to get it:

Slice

```
interface Name
{
    string getName();
    void setName(string name);
}
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

C++

```
p2->setName(p1->getName());
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`. There are no memory-management or exception safety issues with this code.

Exception Handling in C++

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user

exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as C++ exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

C++

```
ChildPrx child = ...;           // Get proxy...
try
{
    child->askToCleanUp();      // Give it a try...
}
catch(const Tantrum& t)
{
    cout << "The child says: " << t.reason << endl;
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

C++

```
int
main(int argc, char* argv[])
{
    int status = 1;
    try
    {
        ChildPrx child = ...;       // Get proxy...
        try
        {
            child->askToCleanUp();  // Give it a try...
            child->praise();        // Give positive feedback...
        }
        catch(const Tantrum& t)
        {
            cout << "The child says: " << t.reason << endl;
            child->scold();         // Recover from error...
        }
        status = 0;
    }
    catch(const Ice::LocalException& e)
    {
        cerr << "Unexpected run?time error: " << e << endl;
    }
    // ...
    return status;
}
```

For efficiency reasons, you should always catch exceptions by `const` reference. This permits the compiler to avoid calling the exception's copy constructor (and, of course, prevents the exception from being sliced to a base type).

Exceptions and Out-Parameters in C++

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may have still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

> This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

Exceptions and Return Values in C++

For return values, C++ provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown. (Of course, this guarantee holds only if you do not use the same variable as both an out-parameter and to receive the return value of an invocation).

See Also

- Operations
- Slice for a Simple File System
- C++98 Mapping for Interfaces

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.

C++98 Mapping for Optional Values

On this page:

The `IceUtil::Optional` Template

The C++ mapping uses a template to hold the values of optional data members and parameters:

C++

```
namespace IceUtil
{
    template<typename T>
    class Optional
    {
    public:
        typedef T element_type;

        Optional();
        Optional(NoneType);
        Optional(const T&);

        template<typename Y>
        Optional(const Optional<Y>&);

        Optional(const Optional& r);

        Optional& operator=(NoneType);
        Optional& operator=(const T&);

        template<typename Y>
        Optional& operator=(const Optional<Y>&);
        Optional& operator=(const Optional&);

        const T& get() const;
        T& get();
        const T* operator->() const;
        T* operator->();
        const T& operator*() const;
        T& operator*();

        operator bool() const;
        bool operator!() const;

        void swap(Optional& other);

        ...
    };
}
```

The `IceUtil::Optional` template provides constructors and assignment operators that allow you to initialize an instance using the element type or an existing optional value. The default constructor initializes an instance to an unset condition. The `get` method and dereference operators retrieve the current value held by the instance, or throw `IceUtil::OptionalNotSetException` if no value is currently set. Use the `bool` or `!` operators to test whether the instance has a value prior to dereferencing it. Finally, the `swap` method exchanges the state of two instances.

The `IceUtil::None` Value

The template includes a constructor and assignment operator that accept `NoneType`. Ice defines an instance of this type, `IceUtil::None`, that you can use to initialize (or reset) an `Optional` instance to an unset condition:

C++

```
IceUtil::Optional<int> i = 5;
i = IceUtil::None;
assert(!i); // true
```

You can pass `IceUtil::None` anywhere an `IceUtil::Optional` value is expected.

See Also

- Optional Data Members
- Operations
- C++98 Mapping for Operations

C++98 Mapping for Classes

On this page:

Basic C++ Mapping for Classes

A Slice class is mapped to a C++ class with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions), and a virtual member function for each operation. Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
}
```

The Slice compiler generates the following code for this definition:

C++

```
class TimeOfDay;

typedef IceInternal::ProxyHandle<IceProxy::TimeOfDay> TimeOfDayPrx;
typedef IceInternal::Handle<TimeOfDay> TimeOfDayPtr;

class TimeOfDay : public virtual Ice::Object
{
public:
    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;

    virtual std::string format() = 0;

    TimeOfDay() {};
    TimeOfDay(Ice::Short, Ice::Short, Ice::Short);

    virtual bool ice_isA(const std::string&);
    virtual const std::string& ice_id();
    static const std::string& ice_staticId();

    typedef TimeOfDayPrx ProxyType;
    typedef TimeOfDayPtr PointerType;

    // ...
};
```

The ProxyType and PointerType definitions are for template programming.

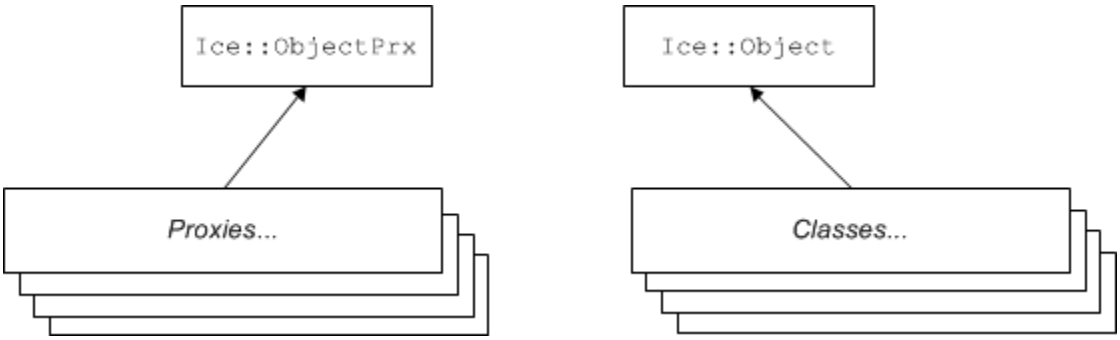
There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice::Object`. This means that all classes implicitly inherit from `Ice::Object`, which is the ultimate ancestor of all classes. Note that `Ice::Object` is *not* the same as `IceProxy::Ice::Object`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The generated class contains a public member for each Slice data member.
3. The generated class has a constructor that takes one argument for each data member, as well as a default constructor.
4. The generated class contains a pure virtual member function for each Slice operation.
5. The generated class contains additional member functions: `ice_isA`, `ice_id`, `ice_staticId`, and `ice_factory`.
6. The compiler generates a type definition `TimeOfDayPtr`. This type implements a smart pointer that wraps dynamically-allocated instances of the class. In general, the name of this type is <*class-name*>`Ptr`. Do not confuse this with <*class-name*>`Prx` — that type exists as well, but is the proxy handle for the class, not a smart pointer.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice::Object` in C++

Like interfaces, classes implicitly inherit from a common C++ base class, `Ice::Object`. However, as shown in the figure below, classes inherited from `Ice::Object` instead of `Ice::ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.



Inheritance from `Ice::ObjectPrx` and `Ice::Object`.

`Ice::Object` contains a number of member functions:

C++

```
namespace Ice
{
    class Object : public virtual IceUtil::Shared
    {
    public:
        virtual bool ice_isA(const std::string&,
const Current& = emptyCurrent) const;
        virtual void ice_ping(const Current& = emptyCurrent) const;
        virtual std::vector<std::string> ice_ids(const Current& = empty
Current) const;
        virtual const std::string& ice_id(const Current& = emptyCurrent
) const;
        static const std::string& ice_staticId();
        virtual ObjectPtr ice_clone() const;

        virtual void ice_preMarshal();
        virtual void ice_postUnmarshal();
        virtual Ice::SlicedDataPtr ice_getSlicedData() const;

        virtual void ice_collectable(bool);

        virtual DispatchStatus ice_dispatch(Ice::Request&,

const DispatchInterceptorAsyncCallbackPtr& = 0);

        virtual bool operator==(const Object&) const;
        virtual bool operator<(const Object&) const;
    };
}
```

The member functions of `Ice::Object` behave as follows:

- `ice_isA`
  This function returns `true` if the object supports the given type ID, and `false` otherwise.
- `ice_ping`

As for interfaces, `ice_ping` provides a basic reachability test for the class. Note that `ice_ping` is normally only invoked on the proxy for a class that might be remote because a class instance that is local (in the caller's address space) can always be reached.

- `ice_ids`
  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This function returns the actual run-time type ID for a class. If you call `ice_id` through a smart pointer to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_staticId`
  This function returns the static type ID of a class.
- `ice_clone`
  This function makes a polymorphic shallow copy of a class.
- `ice_preMarshal`
  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `0` otherwise.
- `ice_collectable`
  Determines whether this object, and by extension the graph of all objects reachable from this object, are eligible for garbage collection when all external references to the graph have been released.
- `ice_dispatch`
  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.
- `operator==`
  `operator<`
  The comparison operators permit you to use classes as elements of STL sorted containers. Note that sort order, unlike for structures, is based on the memory address of the class, not on the contents of its data members of the class.

Class Data Members in C++

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member. Optional data members are mapped to instances of the `IceUtil::Optional` template.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
}
```

The Slice compiler produces the following generated code for this definition:

C++

```
class TimeOfDay : public virtual Ice::Object
{
public:

    virtual std::string format() = 0;

    // ...

protected:

    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;
};
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
}
```

Class Constructors in C++

Classes have a default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value. Optional data members are unset unless they declare default values.

Classes also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement. For each optional data member, its corresponding constructor parameter uses the same mapping as for operation parameters, allowing you to pass its initial value or `IceUtil::None` to indicate an unset value.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order. For example:

Slice

```
class Base
{
    int i;
}

class Derived extends Base
{
    string s;
}
```

This generates:

C++

```
class Base : public virtual ::Ice::Object
{
public:
    ::Ice::Int i;

    Base() {};
    explicit Base(::Ice::Int);

    // ...
};

class Derived : public Base
{
public:
    ::std::string s;

    Derived() {};
    Derived(::Ice::Int, const ::std::string&);

    // ...
};
```

Note that single-parameter constructors are defined as `explicit`, to prevent implicit argument conversions.

By default, derived classes derive non-virtually from their base class. If you need virtual inheritance, you can enable it using the `["cpp:virtual"]` metadata directive.

Class with Operations in C++

> **Deprecated Feature**
> Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

Operations of classes are mapped to pure virtual member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

C++

```
class TimeOfDayI : public virtual TimeOfDay
{
public:
    virtual std::string format()
    {
        std::ostringstream s;
        s << setw(2) << setfill('0') << hour << ":";
        s << setw(2) << setfill('0') << minute << ":";
        s << setw(2) << setfill('0') << second;
        return s.c_str();
    }
};
```

Value Factories in C++

Value factories may be used for classes with or without operations and are *not* deprecated.

Having created a class such as `FormattedTimeOfDayI`, we have an implementation and we can instantiate the `FormattedTimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

Slice

```
interface Time
{
    FormattedTimeOfDay get();
}
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `FormattedTimeOfDay` class. However, `FormattedTimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `FormattedTimeOfDayI` class that implements the abstract `format` operation of the `FormattedTimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `FormattedTimeOfDay` abstract class has a `FormattedTimeOfDayI` concrete implementation.

To supply the Ice run time with a factory for our `FormattedTimeOfDayI` class, we must supply a value factory implementation. The factory's `create` operation is called by the Ice run time when it needs to instantiate a `FormattedTimeOfDay` class. A possible implementation of our value factory is:

C++

```
class ValueFactory : public Ice::ValueFactory
{
public:
    virtual Ice::ObjectPtr create(const std::string& type)
    {
        assert(type == FormattedTimeOfDay::ice_staticId());
        return new FormattedTimeOfDayI;
    }
};
```

The `create` method is passed the type ID of the class to instantiate. For our `FormattedTimeOfDay` class, the type ID is "`::`*`Module`*`::FormattedTimeOfDay`". Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `FormattedTimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoValueFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ValueFactory`, we must inform the Ice run time of the existence of the factory:

C++

```
Ice::CommunicatorPtr ic = ...;
ic->getValueFactoryManager()->add(new ValueFactory,
FormattedTimeOfDay::ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID "`::`*`Module`*`::FormattedTimeOfDay`", it calls the `create` method of the registered `ValueFactory` instance.

Finally, keep in mind that if a class has only data members, but no operations, you do not need to create and register a value factory to receive instances of such a class. You're only required to register a value factory when a class has operations.

See Also

- Classes
- Smart Pointers for Classes
- C++98 Mapping for Operations
- C++98 Mapping for Optional Values
- Asynchronous Method Invocation (AMI) in C++98
- Dispatch Interceptors

Smart Pointers for Classes

On this page:

Automatic Memory Management with Smart Pointers

A recurring theme for C++ programmers is the need to deal with memory allocations and deallocations in their programs. The difficulty of doing so is well known: in the face of exceptions, multiple return paths from functions, and callee-allocated memory that must be deallocated by the caller, it can be extremely difficult to ensure that a program does not leak resources. This is particularly important in multi-threaded programs: if you do not rigorously track ownership of dynamic memory, a thread may delete memory that is still used by another thread, usually with disastrous consequences.

To alleviate this problem, Ice provides smart pointers for classes. These smart pointers use reference counting to keep track of each class instance and, when the last reference to a class instance disappears, automatically delete the instance.

> Smart pointer classes are an example of the *RAII* (*Resource Acquisition Is Initialization*) idiom [1].

Smart pointers are generated by the Slice compiler for each class type. For a Slice class <*class-name*>, the compiler generates a C++ smart pointer called <*class-name*>`Ptr`. Rather than showing all the details of the generated class, here is the basic usage pattern: whenever you allocate a class instance on the heap, you simply assign the pointer returned from `new` to a smart pointer for the class. Thereafter, memory management is automatic and the class instance is deleted once the last smart pointer for it goes out of scope:

C++

```
{                                       // Open scope
    TimeOfDayPtr tod = new TimeOfDayI;  // Allocate instance
    // Initialize...
    tod->hour = 18;
    tod->minute = 11;
    tod->second = 15;
    // ...
}                                       // No memory leak here!
```

As you can see, you use `operator->` to access the members of the class via its smart pointer. When the `tod` smart pointer goes out of scope, its destructor runs and, in turn, the destructor takes care of calling `delete` on the underlying class instance, so no memory is leaked.

A smart pointer performs reference counting of its underlying class instance:

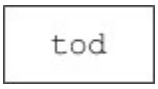
- The constructor of a class sets its reference count to zero.
- Initializing a smart pointer with a dynamically-allocated class instance causes the smart pointer to increment the reference count of the instance by one.
- Copy-constructing a smart pointer increments the reference count of the instance by one.
- Assigning one smart pointer to another increments the target's reference count and decrements the source's reference count. (Self-assignment is safe.)
- The destructor of a smart pointer decrements the reference count by one and calls `delete` on its class instance if the reference count drops to zero.

Suppose that we default-construct a smart pointer as follows:

C++

```
TimeOfDayPtr tod;
```

This creates a smart pointer with an internal null pointer.



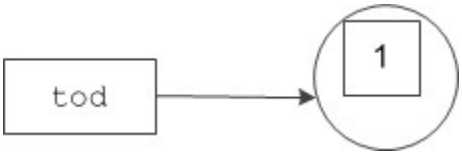
Newly initialized smart pointer.

Constructing a class instance creates that instance with a reference count of zero; the assignment to the smart pointer causes the smart pointer to increment the instance's reference count:

C++

```
tod = new TimeOfDayI;   // Refcount == 1
```

The resulting situation is shown below:



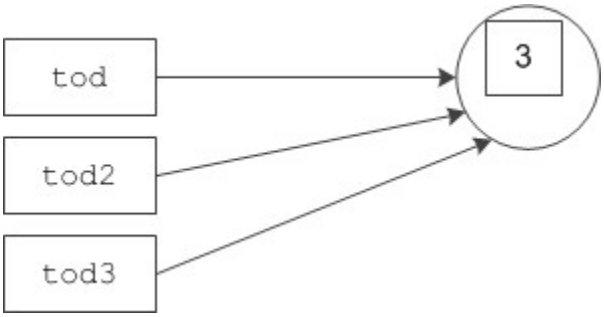
Initialized smart pointer.

Assigning or copy-constructing a smart pointer assigns and copy-constructs the smart pointer (not the underlying instance) and increments the reference count of the instance:

C++

```
TimeOfDayPtr tod2(tod); // Copy-construct tod2
TimeOfDayPtr tod3;
tod3 = tod;             // Assign to tod3
```

Here is the situation after executing these statements:



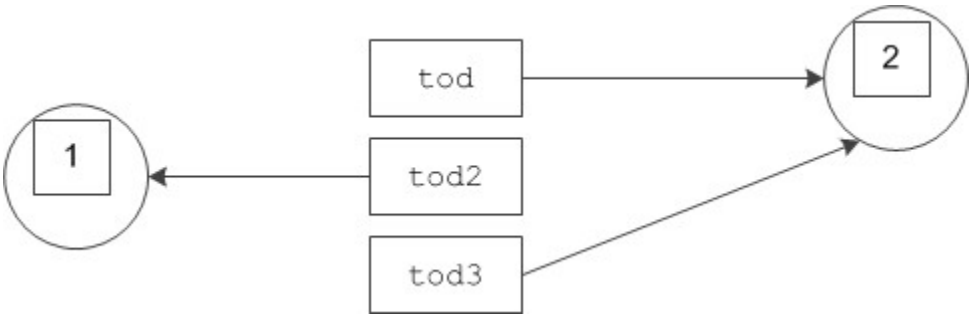
Three smart pointers pointing at the same class instance.

Continuing the example, we can construct a second class instance and assign it to one of the original smart pointers, `tod2`:

C++

```
tod2 = new TimeOfDayI;
```

This decrements the reference count of the instance originally denoted by `tod2` and increments the reference count of the instance that is assigned to `tod2`. The resulting situation becomes the following:



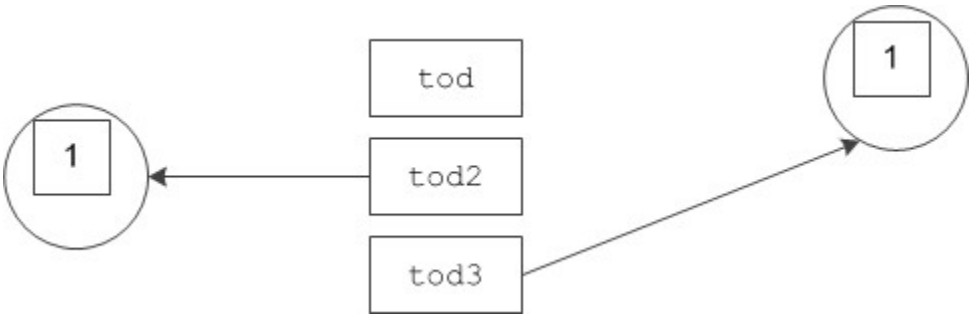
Three smart pointers and two instances.

You can clear a smart pointer by assigning zero to it:

C++

```
tod = 0;        // Clear handle
```

As you would expect, this decrements the reference count of the instance, as shown here:



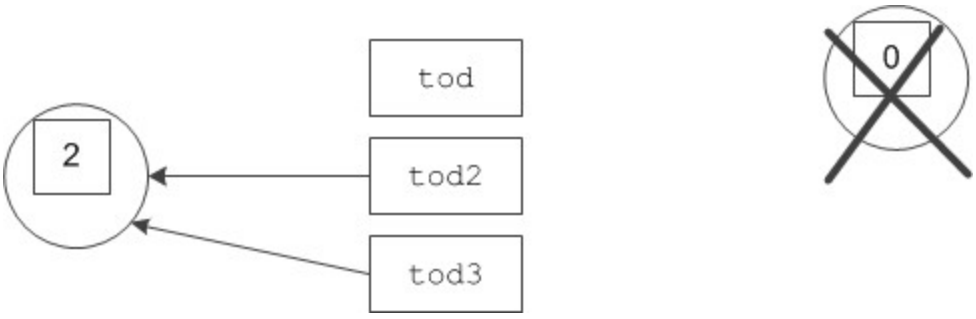
Decremented reference count after clearing a smart pointer.

If a smart pointer goes out of scope, is cleared, or has a new instance assigned to it, the smart pointer decrements the reference count of its instance; if the reference count drops to zero, the smart pointer calls `delete` to deallocate the instance. The following code snippet deallocates the instance on the right by assigning `tod2` to `tod3`:

C++

```
tod3 = tod2;
```

This results in the following situation:



Deallocation of an instance with a reference count of zero.

Copying and Assignment of Classes

Classes have a default memberwise copy constructor and assignment operator, so you can copy and assign class instances:

C++

```
TimeOfDayPtr tod = new TimeOfDayI(2, 3, 4); // Create instance
TimeOfDayPtr tod2 = new TimeOfDayI(*tod);   // Copy instance

TimeOfDayPtr tod3 = new TimeOfDayI;
*tod3 = *tod;                               // Assign instance
```

Copying and assignment in this manner performs a memberwise shallow copy or assignment, that is, the source members are copied into the target members; if a class contains class members (which are mapped as smart pointers), what is copied or assigned is the smart pointer, not the target of the smart pointer.

To illustrate this, consider the following Slice definitions:

Slice

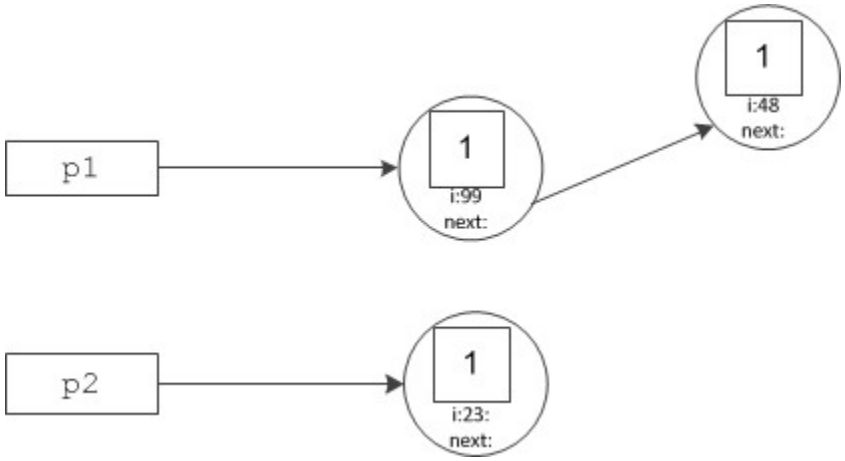
```
class Node
{
    int i;
    Node next;
}
```

Assume that we initialize two instances of type `Node` as follows:

C++

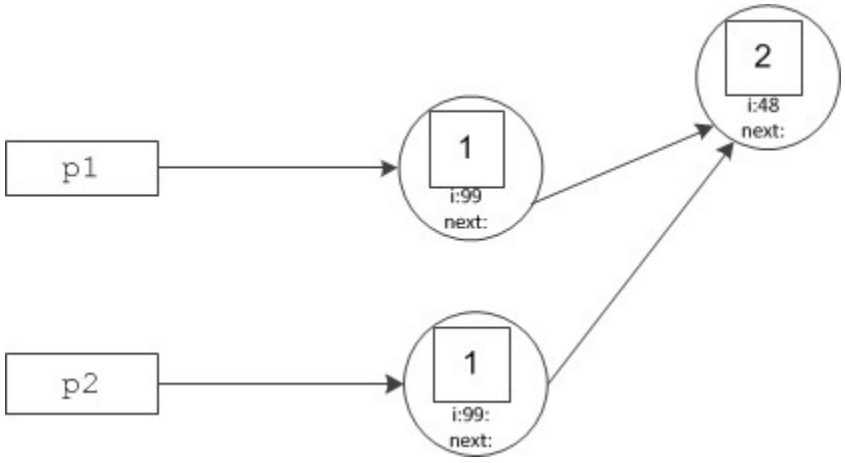
```
NodePtr p1 = new Node(99, new Node(48, 0));
NodePtr p2 = new Node(23, 0);

// ...

*p2 = *p1; // Assignment
```

After executing the first two statements, we have the situation shown below:



Class instances prior to assignment.

After executing the assignment statement, we end up with this result:



Class instances after assignment.

Note that copying and assignment also works for the implementation of abstract classes, such as our `TimeOfDayI` class, for example:

C++

```
class TimeOfDayI;

typedef IceUtil::Handle<TimeOfDayI> TimeOfDayIPtr;

class TimeOfDayI : virtual public TimeOfDay
{
    // As before...
};
```

The default copy constructor and assignment operator will perform a memberwise copy or assignment of your implementation class:

C++

```
TimeOfDayIPtr tod1 = new TimeOfDayI;
TimeOfDayIPtr tod2 = new TimeOfDayI(*tod1);     // Make copy
```

Of course, if your implementation class contains raw pointers (for which a memberwise copy would almost certainly be inappropriate), you must implement your own copy constructor and assignment operator that take the appropriate action (and probably call the base copy constructor or assignment operator to take care of the base part).

Note that the preceding code uses `TimeOfDayIPtr` as a typedef for `IceUtil::Handle<TimeOfDayI>`. This class is a template that contains the smart pointer implementation. If you want smart pointers for the implementation of an abstract class, you must define a smart pointer type as illustrated by this type definition.

Copying and assignment of classes also works correctly for derived classes: you can assign a derived class to a base class, but not vice-versa; during such an assignment, the derived part of the source class is sliced, as per the usual C++ assignment semantics.

Polymorphic Copying of Classes

As shown in Inheritance from `Ice::Object`, the C++ mapping generates an `ice_clone` member function for every class:

C++

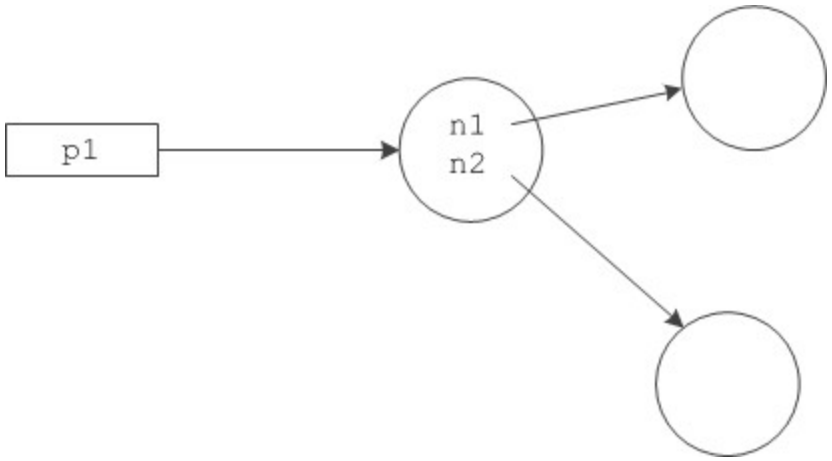
```
class TimeOfDay : public virtual Ice::Object
{
public:
    // ...

    virtual Ice::ObjectPtr ice_clone() const;
};
```

This member function makes a polymorphic shallow copy of a class: members that are not class members are deep copied; all members that are class members are shallow copied. To illustrate, consider the following class definition:

Slice

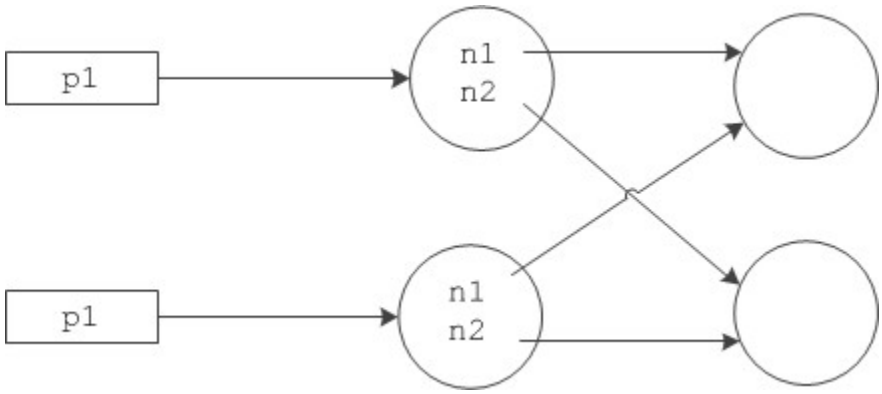
```
class Node
{
    Node n1;
    Node n2;
}
```

Assume that we have an instance of this class, with the `n1` and `n2` members initialized to point at separate instances, as shown below:



A class instance pointing at two other instances.

If we call `ice_clone` on the instance on the left, we end up with this situation:



Resulting graph after calling `ice_clone` *on the left-most instance.*

As you can see, class members are shallow copied, that is, `ice_clone` makes a copy of the class instance on which it is invoked, but does not copy any class instances that are pointed at by the copied instance.

Note that `ice_clone` returns a value of type `Ice::ObjectPtr`, to avoid problems with compilers that do not support covariant return types. The generated `Ptr` classes contain a `dynamicCast` member that allows you to safely down-cast the return value of `ice_clone`. For example, the code to achieve the situation shown in the illustration above, looks as follows:

C++

```
NodePtr p1 = new Node(new Node, new Node);
NodePtr p2 = NodePtr::dynamicCast(p1->ice_clone());
```

`ice_clone` is generated by the Slice compiler for concrete classes (that is, classes that do not have operations). However, because classes with operations are abstract, the generated `ice_clone` for abstract classes cannot know how to instantiate an instance of the derived concrete class (because the name of the derived class is not known). This means that, for abstract classes, the generated `ice_clone` throws a `CloneNotImplementedException`.

If you want to clone the implementation of an abstract class, you must override the virtual `ice_clone` member in your concrete implementation class. For example:

C++

```
class TimeOfDayI : public TimeOfDay
{
public:
    virtual Ice::ObjectPtr ice_clone() const
    {
        return new TimeOfDayI(*this);
    }
};
```

Null Smart Pointers

A null smart pointer contains a null C++ pointer to its underlying instance. This means that if you attempt to dereference a null smart pointer, you get an `IceUtil::NullHandleException`:

C++

```
TimeOfDayPtr tod;               // Construct null handle

try
{
    tod->minute = 0;            // Dereference null handle
    assert(false);              // Cannot get here
}
catch(const IceUtil::NullHandleException&)
{
    // OK, expected
}
catch(...)
{
    assert(false);              // Must get NullHandleException
}
```

Preventing Stack-Allocation of Class Instances

The Ice C++ mapping expects class instances to be allocated on the heap. Allocating class instances on the stack or in static variables is pragmatically useless because all the Ice APIs expect parameters that are smart pointers, not class instances. This means that, to do anything with a stack-allocated class instance, you must initialize a smart pointer for the instance. However, doing so does not work because it inevitably leads to a crash:

C++

```
{                                   // Enter scope
    TimeOfDayI t;                   // Stack-allocated class instance
    TimeOfDayPtr todp;              // Handle for a TimeOfDay instance

    todp = &t;                      // Legal, but dangerous
    // ...
}                                   // Leave scope, looming crash!
```

This goes badly wrong because, when `todp` goes out of scope, it decrements the reference count of the class to zero, which then calls `delete` on itself. However, the instance is stack-allocated and cannot be deleted, and we end up with undefined behavior (typically, a core dump).

The following attempt to fix this is also doomed to failure:

C++

```
{                                   // Enter scope
    TimeOfDayI t;                   // Stack-allocated class instance
    TimeOfDayPtr todp;              // Handle for a TimeOfDay instance

    todp = &t;                      // Legal, but dangerous
    // ...
    todp = 0;                       // Crash imminent!
}
```

This code attempts to circumvent the problem by clearing the smart pointer explicitly. However, doing so also causes the smart pointer to drop the reference count on the class to zero, so this code ends up with the same call to `delete` on the stack-allocated instance as the previous example.

The upshot of all this is: **never allocate a class instance on the stack or in a static variable**. The C++ mapping assumes that all class instances are allocated on the heap and no amount of coding trickery will change this.

You could abuse the `__setNoDelete` member to disable deallocation, but we strongly discourage you from doing this.

You can prevent allocation of class instances on the stack or in static variables by adding a protected destructor to your implementation of the class: if a class has a protected destructor, allocation must be made with `new`, and static or stack allocation causes a compile-time error. In addition, explicit calls to `delete` on a heap-allocated instance also are flagged at compile time.

Tip
You may want to habitually add a protected destructor to your implementation of abstract Slice classes to protect yourself from accidental heap allocation, as shown in Class Operations. (For Slice classes that do not have operations, the Slice compiler automatically adds a protected destructor.)

Smart Pointers and Constructors

Slice classes inherit their reference-counting behavior from the `IceUtil::Shared` class, which ensures that reference counts are managed in a thread-safe manner. When a stack-allocated smart pointer goes out of scope, the smart pointer invokes the `__decRef` function on the reference-counted object. Ignoring thread-safety issues, `__decRef` is implemented like this:

C++

```
void
IceUtil::Shared::__decRef()
{
    if(--_ref == 0 && !_noDelete)
    {
        delete this;
    }
}
```

In other words, when the smart pointer calls `__decRef` on the pointed-at instance and the reference count reaches zero (which happens when the last smart pointer for a class instance goes out of scope), the instance self-destructs by calling `delete this`.

However, as you can see, the instance self-destructs only if `_noDelete` is false (which it is by default, because the constructor initializes it to false). You can call `__setNoDelete(true)` to prevent this self-destruction and, later, call `__setNoDelete(false)` to enable it again. This is necessary if, for example, a class in its constructor needs to pass `this` to another function:

C++

```
void someFunction(const TimeOfDayPtr& t)
{
    // ...
}

TimeOfDayI::TimeOfDayI()
{
    someFunction(this); // Trouble looming here!
}
```

At first glance, this code looks innocuous enough. While `TimeOfDayI` is being constructed, it passes `this` to `someFunction`, which expects a smart pointer. The compiler constructs a temporary smart pointer at the point of call (because the smart pointer template has a single-argument constructor that accepts a pointer to a heap-allocated instance, so the constructor acts as a conversion function). However, this code fails badly. The `TimeOfDayI` instance is constructed with a statement such as:

C++

```
TimeOfDayPtr tod = new TimeOfDayI;
```

The constructor of `TimeOfDayI` is called by `operator new` and, when the constructor starts executing, the reference count of the instance is zero (because that is what the reference count is initialized to by the `Shared` base class of `TimeOfDayI`). When the constructor calls `someFunction`, the compiler creates a temporary smart pointer, which increments the reference count of the instance and, once `someFunction` completes, the compiler dutifully destroys that temporary smart pointer again. But, of course, that drops the reference count back to zero and causes the `TimeOfDayI` instance to self-destruct by calling `delete this`. The net effect is that the call to `new TimeOfDayI` returns a pointer to an already deleted object, which is likely to cause the program to crash.

To get around the problem, you can call `__setNoDelete`:

C++

```
TimeOfDayI::TimeOfDayI()
{
    __setNoDelete(true);
    someFunction(this);
    __setNoDelete(false);
}
```

The code disables self-destruction while `someFunction` uses its temporary smart pointer by calling `__setNoDelete(true)`. By doing this, the reference count of the instance is incremented before `someFunction` is called and decremented back to zero when `someFunction` completes without causing the object to self-destruct. The constructor then re-enables self-destruction by calling `__setNoDelete(false)` before returning, so the statement

C++

```
TimeOfDayPtr tod = new TimeOfDayI;
```

does the usual thing, namely to increment the reference count of the object to 1, despite the fact that a temporary smart pointer existed while the constructor ran.

> In general, whenever a class constructor passes `this` to a function or another class that accepts a smart pointer, you must temporarily disable self-destruction.

Smart Pointers and Exception Safety

Smart pointers are exception safe: if an exception causes the thread of execution to leave a scope containing a stack-allocated smart pointer, the C++ run time ensures that the smart pointer's destructor is called, so no resource leaks can occur:

C++

```
{ // Enter scope...

    TimeOfDayPtr tod = new TimeOfDayI; // Allocate instance

    someFuncThatMightThrow();          // Might throw...

    // ...

} // No leak here, even if an exception is thrown
```

If an exception is thrown, the destructor of `tod` runs and ensures that it deallocates the underlying class instance.

There is one potential pitfall you must be aware of though: if a constructor of a base class throws an exception, and another class instance holds a smart pointer to the instance being constructed, you can end up with a double deallocation. You can use the `__setNoDelete` mechanism to temporarily disable self-destruction in this case, as described above.

Smart Pointers and Cycles

One thing you need to be aware of is the inability of reference counting to deal with cyclic dependencies. For example, consider the following

simple self-referential class:

Slice

```
class Node
{
    int val;
    Node next;
}
```

Intuitively, this class implements a linked list of nodes. As long as there are no cycles in the list of nodes, everything is fine, and our smart pointers will correctly deallocate the class instances. However, if we introduce a cycle, we have a problem:

C++

```
{                           // Open scope...

    NodePtr n1 = new Node; // N1 refcount == 1
    NodePtr n2 = new Node; // N2 refcount == 1
    n1->next = n2;         // N2 refcount == 2
    n2->next = n1;         // N1 refcount == 2

} // Destructors run:      // N2 refcount == 1,
                           // N1 refcount == 1, memory leak!
```

The nodes pointed to by `n1` and `n2` do not have names but, for the sake of illustration, let us assume that `n1`'s node is called N1, and `n2`'s node is called N2. When we allocate the N1 instance and assign it to `n1`, the smart pointer `n1` increments N1's reference count to 1. Similarly, N2's reference count is 1 after allocating the node and assigning it to `n2`. The next two statements set up a cyclic dependency between `n1` and `n2` by making their `next` pointers point at each other. This sets the reference count of both N1 and N2 to 2. When the enclosing scope closes, the destructor of `n2` is called first and decrements N2's reference count to 1, followed by the destructor of `n1`, which decrements N1's reference count to 1. The net effect is that neither reference count ever drops to zero, so both N1 and N2 are leaked.

Garbage Collection of Class Instances

The previous example illustrates a problem that is generic to using reference counts for deallocation: if a cyclic dependency exists anywhere in a graph (possibly via many intermediate nodes), all nodes in the cycle are leaked.

To avoid memory leaks due to such cycles, Ice for C++ includes a garbage collection facility. The facility identifies class instances that are part of a cycle but are no longer reachable from the program and deletes such instances. Applications must assist the Ice run time in this process by indicating when a graph is eligible for collection. For eligible graphs, Ice makes a sweep of the graph each time a reference count to one of the graph's objects is decremented.

Two components of the garbage collection facility influence its behavior:

- The `Ice.CollectObjects` property determines whether Ice assumes all object graphs are eligible for collection by default.
- The `Object::ice_collectable(bool)` method allows an application to indicate whether an object (and by extension, the graph of objects reachable via this object) is eligible for collection.

> The correct operation of the garbage collection facility relies on the assumption that all eligible object graphs are immutable. If an application needs to make changes that could affect the structure of the graph, it must disable collection for that graph by calling `ice_collectable(false)` on any root object in the graph. Once the changes are complete, call `ice_collectable(true)` on any root object in the graph to make it eligible again. Modifying the structure of an eligible graph has undefined behavior.

In general, there are two strategies you can use for garbage collection depending on your application's requirements:

1. Set `Ice.CollectObjects=1` so that Ice assumes all object graphs are eligible for collection by default. This is recommended for applications that receive object graphs but rarely modify them. For those situations where an application needs to modify a graph, surround the modification with calls to `ice_collectable` as shown below:

C++

```
NodePtr graph = proxy->getGraph(); // Eligible by default
graph->ice_collectable(false);
// modify graph...
graph->ice_collectable(true);
graph = 0; // Starts a sweep
```

2. Set `Ice.CollectObjects=0` (the default setting) so that Ice does not attempt to collect object graphs except for those explicitly marked by the application. Use this setting for applications that typically modify the structure of the graphs they receive. Call `ice_collectable(true)` to mark a graph as eligible:

C++

```
NodePtr graph = proxy->getGraph(); // Ineligible by default
// modify graph...
graph->ice_collectable(true);
graph = 0; // Starts a sweep
```

As mentioned earlier, an application does not take any action to force a collection; rather, the collection occurs automatically when a reference count is decremented.

> To minimize overhead, GC-related behavior is only enabled for those Slice classes whose data members can refer to Ice objects. Furthermore, graph traversal only occurs for those objects that are part of a cycle and marked as collectable.

Smart Pointer Comparison

As for proxy handles, class handles support the comparison operators ==, !=, and <. This allows you to use class handles in STL sorted containers. Be aware that, for smart pointers, object identity is not used for the comparison, because class instances do not have identity. Instead, these operators simply compare the memory address of the classes they point to. This means that `operator==` returns true only if two smart pointers point at the same physical class instance:

C++

```
// Create a class instance and initialize
//
TimeOfDayIPtr p1 = new TimeOfDayI;
p1->hour = 23;
p1->minute = 10;
p1->second = 18;

// Create another class instance with
// the same member values
//
TimeOfDayIPtr p2 = new TimeOfDayI;
p2->hour = 23;
p2->minute = 10;
p2->second = 18;

assert(p1 != p2);       // The two do not compare equal

TimeOfDayIPtr p3 = p1;  // Point at first class again

assert(p1 == p3);       // Now they compare equal
```

See Also

- Classes
- C++98 Mapping for Classes
- Asynchronous Method Invocation (AMI) in C++98
- Application Helper Class
- Properties and Configuration
- The C++ Shared and SimpleShared Classes

References

1. Stroustrup, B. 1997. *The C++ Programming Language*. Reading, MA: Addison-Wesley.

Asynchronous Method Invocation (AMI) in C++98

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

Basic Asynchronous API in C++

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in C++

Besides the synchronous proxy methods, `slice2cpp` generates the following asynchronous proxy methods:

C++

```
Ice::AsyncResultPtr begin_getName(Ice::Int number);
Ice::AsyncResultPtr begin_getName(Ice::Int number,
const Ice::Context& __ctx)

std::string end_getName(const Ice::AsyncResultPtr&);
```

Four additional overloads of `begin_getName` are generated for use with generic callbacks and type-safe callbacks.

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. (The `begin_` method is overloaded

so you can pass a per-invocation context.)

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

C++

```
EmployeesPrx e = ...;
Ice::AsyncResultPtr r = e->begin_getName(99);

// Continue to do other things here...

string name = e->end_getName(r);
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `AsyncResultPtr`. The `AsyncResult` associated with this smart pointer contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResultPtr` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. Similarly, the `end_` method has one out-parameter for each out-parameter of the corresponding Slice operation (plus the `AsyncResultPtr` parameter). For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

C++

```
Ice::AsyncResultPtr begin_op(Ice::Int inp1, const ::std::string& inp2)

Ice::Double end_op(bool& outp1, Ice::Long& outp2,
const Ice::AsyncResultPtr&);
```

Asynchronous Exception Semantics in C++

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` meth od is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `IceUtil::IllegalArgumentException`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a

return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

`AsyncResult` Class in C++

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

C++

```
class AsyncResult : public Ice::LocalObject,
private IceUtil::noncopyable
{
public:
    bool operator==(const AsyncResult&) const;
    bool operator<(const AsyncResult&) const;

    Int getHash() const;

    void cancel();

    CommunicatorPtr getCommunicator() const;
    ConnectionPtr getConnection() const;
    ObjectPrx getProxy() const;
    const string& getOperation() const;
    LocalObjectPtr getCookie() const;

    bool isCompleted() const;
    void waitForCompleted();

    bool isSent() const;
    void waitForSent();

    void throwLocalException() const;

    bool sentSynchronously() const;
};
```

The methods have the following semantics:

- `bool operator==(const AsyncResult&) const`
  `bool operator<(const AsyncResult&) const`
  `Int getHash() const`
  These methods allow you to create ordered or hashed collections of pending asynchronous invocations. This is useful, for example, if you can have a number of outstanding requests, and need to pass state between the `begin_` and the `end_` methods. In this case, you can use the returned `AsyncResult` objects as keys into a map that stores the state for each call.
- `void cancel()`
  This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. `cancel` is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning `isCompleted` returns true, and the result of the invocation is an `Ice::InvocationCanceledException`.
- `CommunicatorPtr getCommunicator() const`
  This method returns the communicator that sent the invocation.

- `virtual ConnectionPtr getConnection() const`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `getConnection` method only returns a non-nil value when the `AsyncResult` object is obtained by calling `begin_flushBatchRequests` on a `Connection` object.
- `virtual ObjectPrx getProxy() const`
  This method returns the proxy that was used to call the `begin_` method, or nil if the `AsyncResult` object was not obtained via an asynchronous proxy invocation.
- `const string& getOperation() const`
  This method returns the name of the operation.
- `LocalObjectPtr getCookie() const`
  This method returns the cookie that was passed to the `begin_` method. If you did not pass a cookie to the `begin_` method, the return value is null.
- `bool isCompleted() const`
  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.
- `void waitForCompleted()`
  This method blocks the caller until the result of an invocation becomes available.
- `bool isSent() const`
  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.
- `void waitForSent()`
  This method blocks the calling thread until a request has been written to the client-side transport, or an exception occurs. After `waitForSent` returns, `isSent` returns true if the request was successfully written to the client-side transport, or false if an exception occurred. In the case of a failure, you can call the corresponding `end_` method or `throwLocalException` to obtain the exception.
- `void throwLocalException() const`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.
- `bool sentSynchronously() const`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

Polling for Completion in C++

The `AsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

C++

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;

Ice::Int offset = 0;
while(!file.eof())
{
    ByteSeq bs;
    bs = file.read(chunkSize); // Read a chunk
    ft->send(offset, bs);      // Send the chunk
    offset += bs.size();
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

C++

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
Ice::Int offset = 0;

list<Ice::AsyncResultPtr> results;
const int numRequests = 5;

while(!file.eof())
{
    ByteSeq bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice::AsyncResultPtr r = ft->begin_send(offset, bs);
    offset += bs.size();

    // Wait until this request has been passed to the transport.
    r->waitForSent();
    results.push_back(r);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.size() > numRequests)
    {
        Ice::AsyncResultPtr r = results.front();
        results.pop_front();
        r->waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while(!results.empty())
{
    Ice::AsyncResultPtr r = results.front();
    results.pop_front();
    r->waitForCompleted();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Generic Completion Callbacks in C++

The `begin_` method is overloaded to allow you to provide completion callbacks. Here are the corresponding methods for the `getName` operation:

C++

```
Ice::AsyncResultPtr begin_getName(
                        Ice::Int number,
                        const Ice::CallbackPtr& __del,
                        const Ice::LocalObjectPtr& __cookie = 0);

Ice::AsyncResultPtr begin_getName(
                        Ice::Int number,
                        const Ice::Context& __ctx,
                        const Ice::CallbackPtr& __del,
                        const Ice::LocalObjectPtr& __cookie = 0);
```

The second version of `begin_getName` lets you override the default context. (We discuss the purpose of the `cookie` parameter in the next section.) Following the in-parameters, the `begin_` method accepts a parameter of type `Ice::CallbackPtr`. This is a smart pointer to a callback class that is provided by the Ice run time. This class stores an instance of a callback class that you implement. The Ice run time invokes a method on your callback instance when an asynchronous operation completes. Your callback class must provide a method that returns `void` and accepts a single parameter of type `const AsyncResultPtr&`, for example:

C++

```
class MyCallback : public IceUtil::Shared
{
public:
    void finished(const Ice::AsyncResultPtr& r)
    {
        EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
        try
        {
            string name = e->end_getName(r);
            cout << "Name is: " << name << endl;
        }
        catch(const Ice::Exception& ex)
        {
            cerr << "Exception is: " << ex << endl;
        }
    }
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

Note that your callback class must derive from `IceUtil::Shared`. The callback method can have any name you prefer but its signature must match the preceding example.

The implementation of your callback method must call the `end_` method. The proxy for the call is available via the `getProxy` method on the `AsyncResult` that is passed by the Ice run time. The return type of `getProxy` is `Ice::ObjectPrx`, so you must down-cast the proxy to

its correct type. (You should always use an `uncheckedCast` to do this, otherwise you will send an additional message to the server to verify the proxy type.)

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. If you allow an exception to escape from the callback method, the Ice run time produces a log entry by default and ignores the exception. (You can disable the log message by setting the property `Ice.Warn.AMICallback` to zero.)

To inform the Ice run time that you want to receive a callback for the completion of the asynchronous call, you pass the callback instance to the `begin_` method:

C++

```
EmployeesPrx e = ...;

MyCallbackPtr cb = new MyCallback;
Ice::CallbackPtr d = Ice::newCallback(cb, &MyCallback::finished);

e->begin_getName(99, d);
```

Note the call to `Ice::newCallback` in this example. This helper function expects a smart pointer to your callback instance and a member function pointer that specifies your callback method.

Using Cookies for Generic Completion Callbacks in C++

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

The API allows you to pass such state by providing a cookie. A cookie is an instance of a class that you write; the class can contain whatever data you want to pass, as well as any methods you may want to add to manipulate that data.

The only requirement on the cookie class is that it must derive from `Ice::LocalObject`. Here is an example implementation that stores a `WidgetHandle`. (We assume that this class provides whatever methods are needed by the `end_` method to update the display.)

C++

```
class Cookie : public Ice::LocalObject
{
public:
    Cookie(WidgetHandle h) : _h(h) {}
    WidgetHandle getWidget() { return _h; }

private:
    WidgetHandle _h;
};
typedef IceUtil::Handle<Cookie> CookiePtr;
```

When you call the `begin_` method, you pass the appropriate cookie instance to inform the `end_` method how to update the display:

C++

```
// Make cookie for call to getName(99).
CookiePtr cookie1 = new Cookie(widgetHandle1);

// Make cookie for call to getName(42);
CookiePtr cookie2 = new Cookie(widgetHandle2);

// Invoke the getName operation with different cookies.
e->begin_getName(99, getNameCB, cookie1);
e->begin_getName(24, getNameCB, cookie2);
```

The `end_` method can retrieve the cookie from the `AsyncResult` by calling `getCookie`. For this example, we assume that widgets have a `writeString` method that updates the relevant UI element:

C++

```
void
MyCallback::getName(const Ice::AsyncResultPtr& r)
{
    EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
    CookiePtr cookie = CookiePtr::dynamicCast(r->getCookie());
    try
    {
        string name = e->end_getName(r);
        cookie->getWidget()->writeString(name);
    }
    catch(const Ice::Exception& ex)
    {
        handleException(ex);
    }
}
```

The cookie provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same cookie instance to multiple invocations.

Type-Safe Completion Callbacks in C++

The generic callback API is not entirely type-safe:

- You must down-cast the return value of `getProxy` to the correct proxy type before you can call the `end_` method.
- You must call the correct `end_` method to match the operation called by the `begin_` method.
- If you use a cookie, you must down-cast the cookie to the correct type before you can access the data inside the cookie.
- You must remember to catch exceptions when you call the `end_` method; if you forget to do this, you will not know that the operation failed.

`slice2cpp` generates an additional type-safe API that takes care of these chores for you. The type-safe API is provided as a template that works much like the `Ice::Callback` class of the generic API, but requires strongly-typed method signatures.

To use type-safe callbacks, you must implement a callback class that provides two callback methods:

- A success callback that is called if the operation succeeds
- A failure callback that is called if the operation raises an exception

As for the generic API, your callback class must derive from `IceUtil::Shared`. Here is a callback class for an invocation of the `getName` operation:

C++

```
class MyCallback : public IceUtil::Shared
{
public:
    void getNameCB(const string& name)
    {
        cout << "Name is: " << name << endl;
    }

    void failureCB(const Ice::Exception& ex)
    {
        cerr << "Exception is: << ex << endl;
    }
};
```

The callback methods can have any name you prefer and must have `void` return type. The failure callback always has a single parameter of type `const Ice::Exception&`. The success callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the success callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

To receive these callbacks, you instantiate your callback instance and specify the methods you have defined before passing a smart pointer to a callback wrapper instance to the `begin_` method:

C++

```
MyCallbackPtr cb = new MyCallback;

Callback_Employees_getNamePtr getNameCB =
    newCallback_Employees_getName(cb, &MyCallback::getNameCB, &MyCallba
ck::failureCB);

Callback_Employees_getNumberPtr getNumberCB =
    newCallback_Employees_getNumber(cb, &MyCallback::getNumberCB, &MyCa
llback::failureCB);

e->begin_getName(99, getNameCB);
e->begin_getNumber("Fred", getNumberCB);
```

Note how this code creates instances of two smart pointer types generated by `slice2cpp` named `Callback_Employees_getNamePtr` and `Callback_Employees_getNumberPtr`. Each smart pointer points to a template instance that encapsulates your callback instance and two member function pointers for the callback methods. The name of this smart pointer type is formed as follows:

<module>`::Callback_<`*interface*`>_<`*operation*`>Ptr`

Also note that the code uses helper functions to initialize the smart pointers. The first argument to the helper function is your callback instance, and the two following arguments are the success and failure member function pointers, respectively. The name of this helper function is formed as follows:

`<module>::newCallback_<interface>_<operation>`

It is legal to pass a null pointer as the success or failure callback. For the success callback, this is legal only for operations that have `void` return type and no out-parameters. This is useful if you do not care when the operation completes but want to know if the call failed. If you pass a null exception callback, the Ice run time will ignore any exception that is raised by the invocation.

The type of the success and exception member function pointers is provided as `Response` and `Exception` typedefs by the callback template. For example, you can ignore exceptions for an invocation of `getName` as follows:

C++

```
Callback_Employees_getName::Exception nullException = 0;

MyCallbackPtr cb = new MyCallback;

Callback_Employees_getNamePtr getNameCB =
    newCallback_Employees_getName(cb, &MyCallback::getNameCB, nullExcep
tion);

e->begin_getName(99, getNameCB); // Ignores exceptions
```

Using Cookies for Type-Safe Completion Callbacks in C++

The `begin_` method optionally accepts a cookie as a trailing parameter. As for the generic API, you can use the cookie to share state between the `begin_` and `end_` methods. However, with the type-safe API, there is no need to down-cast the cookie. Instead, the cookie parameter that is passed to the `end_` method is strongly typed. Assuming that you have defined a `Cookie` class and `CookiePtr` smart pointer, you can pass a cookie to the `begin_` method as follows:

C++

```
MyCallbackPtr cb = new MyCallback;

Callback_Employees_getNamePtr getNameCB =
    newCallback_Employees_getName(cb, &MyCallback::getNameCB, &MyCallba
ck::failureCB);

CookiePtr cookie = new Cookie(widgetHandle);
e->begin_getName(99, getNameCB, cookie);
```

The callback methods of your callback class simply add the cookie parameter:

C++

```
class MyCallback : public IceUtil::Shared
{
public:
    void getNameCB(const string& name, const CookiePtr& cookie)
    {
        cookie->getWidget()->writeString(name);
    }

    void failureCB(const Ice::Exception& ex, const CookiePtr& cookie)
    {
        cookie->getWidget()->writeError(ex.what());
    }
};
```

Asynchronous Oneway Invocations in C++

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws an `IceUtil::IllegalArgumentException`.

For the generic API, the callback method looks exactly as for a twoway invocation. However, for oneway invocations, the Ice run time does not call the callback method unless the invocation raised an exception during the `begin_` method ("on the way out").

For the type-safe API, the `newCallback` helper for `void` operations is overloaded so you can omit the success callback. For example, here is how you could call `ice_ping` asynchronously:

C++

```
MyCallbackPtr cb = new MyCallback;

Ice::Callback_Object_ice_pingPtr callback =
    Ice::newCallback_Object_ice_ping(cb, &MyCallback::failureCB);

p->begin_opVoid(callback);
```

Flow Control in C++

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult::sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult::sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds

some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

For the generic API, you can create an additional callback method:

C++

```
class MyCallback : public IceUtil::Shared
{
public:
    void finished(const Ice::AsyncResultPtr&);
    void sent(const Ice::AsyncResultPtr&);
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

As with any other callback method, you are free to choose any name you like. For this example, the name of the callback method is `sent`. You inform the Ice run time that you want to be informed when a call has been passed to the local transport by specifying the `sent` method as an additional parameter when you create the `Ice::Callback`:

C++

```
EmployeesPrx e = ...;

MyCallbackPtr cb = new MyCallback;
Ice::CallbackPtr d = Ice::newCallback(cb, &MyCallback::finished,
&MyCallback::sent);

e->begin_getName(99, d);
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the `sent` method from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the `sent` method from a different thread once it has written the request to the local transport. In addition, you can find out from the `AsyncResult` that is returned by the `begin_` method whether the request was sent synchronously or was queued, by calling `sentSynchronously`.

For the generic API, the `sent` method has the following signature:

C++

```
void sent(const Ice::AsyncResult&);
```

For the type-safe API, there are two versions, one without and one with a cookie:

C++

```
void sent(bool sentSynchronously);
void sent(bool sentSynchronously, const <CookiePtr>& cookie);
```

For the version with a cookie, *<CookiePtr>* is replaced with the actual type of the cookie smart pointer you passed to the `begin_` method.

The `sent` methods allow you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Asynchronous Batch Requests in C++

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws an `IceUtil::IllegalArgumentException`.

A batch oneway invocation never calls the generic or type-safe callbacks unless an error occurs before the request is queued. The returned `Ice::AsyncResult` for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The returned result can also be marked completed if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult::getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency Semantics for AMI in C++

The Ice run time always invokes your callback methods from a separate thread. This means that you can safely use a non-recursive mutex without risking deadlock. There is one exception to this rule: the run time calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or parameter, so you can take appropriate action to avoid a self-deadlock.

See Also

- C++98 Mapping for Classes
- Smart Pointers for Classes
- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch

Using Slice Checksums in C++98

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2cpp`, the `--checksum` option causes the compiler to generate code in each C++ source file that accumulates checksums in a global map. A copy of this map can be obtained by calling a function defined in the header file `Ice/SliceChecksums.h`:

C++

```
namespace Ice
{
    Ice::SliceChecksumDict sliceChecksums();
}
```

In order to verify a server's checksums, a client could simply compare the dictionaries using the equality operator. However, this is not feasible if it is possible that the server might be linked with more Slice definitions than the client. A more general solution is to iterate over the local checksums as demonstrated below:

C++

```
Ice::SliceChecksumDict serverChecksums = ...
Ice::SliceChecksumDict localChecksums = Ice::sliceChecksums();

for(Ice::SliceChecksumDict::const_iterator p = localChecksums.begin();
p != localChecksums.end(); ++p)
{
    Ice::SliceChecksumDict::const_iterator q
= serverChecksums.find(p->first);
    if(q == serverChecksums.end())
    {
        // No match found for type id!
    }
    else if(p->second != q->second)
    {
        // Checksum mismatch!
    }
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums
- slice2cpp Command-Line Options (C++98)

Example of a File System Client in C++98

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The C++ code shown here hardly differs from the code you would write for an ordinary C++ program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local C++ object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side C++ mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

C++

```
#include <Ice/Ice.h>
#include <Filesystem.h>
#include <iostream>
#include <iterator>
```

```
using namespace std;
using namespace Filesystem;

static void
listRecursive(const DirectoryPrx& dir, int depth = 0)
{
    // ...
}

int
main(int argc, char* argv[])
{
    int status = 0;
    try
    {
        // Create a communicator
        //
        Ice::CommunicatorHolder ich(argc, argv);

        // Create a proxy for the root directory
        //
        Ice::ObjectPrx base
= ich->stringToProxy("RootDir:default -p 10000");
        if(!base)
        {
            throw "Could not create proxy";
        }

        // Down-cast the proxy to a Directory proxy
        //
        DirectoryPrx rootDir = DirectoryPrx::checkedCast(base);
        if(!rootDir)
        {
            throw "Invalid proxy";
        }

        // Recursively list the contents of the root directory
        //
        cout << "Contents of root directory:" << endl;
        listRecursive(rootDir);
    }
    catch(const Ice::Exception& ex)
    {
        cerr << ex << endl;
        status = 1;
    }
    catch (const char* msg)
    {
        cerr << msg << endl;
```

```
        status = 1;
    }
```

```
    return status;
}
```

1. The code includes a few header files:
   - `Ice/Ice.h`:
     Always included in both client and server source files, provides definitions that are necessary for accessing the Ice run time.
   - `Filesystem.h`:
     The header that is generated by the Slice compiler from the Slice definitions in `Filesystem.ice`.
   - `iostream`:
     The client uses the `iostream` library to produce its output.
   - `iterator`:
     The implementation of `listRecursive` uses an STL iterator.
2. The code adds `using` declarations for the `std` and `Filesystem` namespaces.
3. The structure of the code in `main` follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
4. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`:

C++

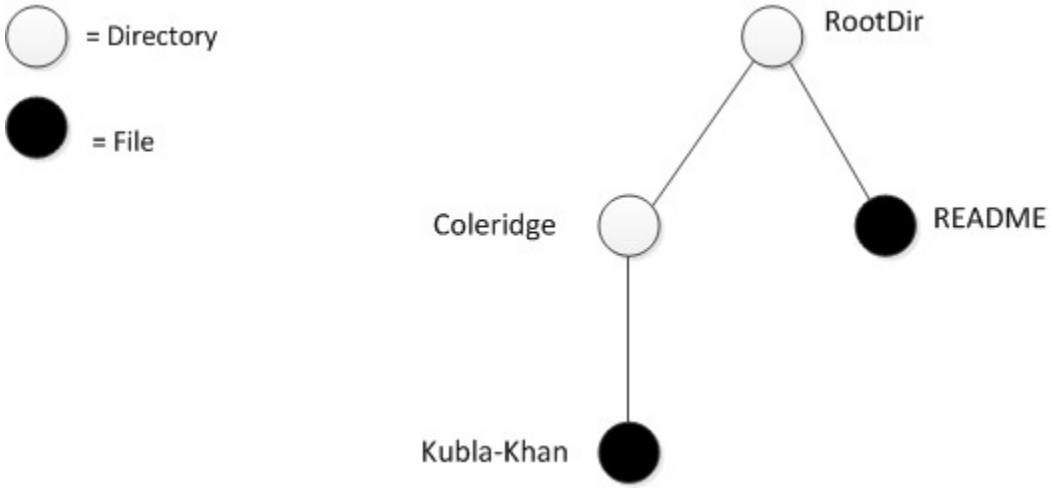
```
// Recursively print the contents of directory "dir" in
// tree fashion. For files, show the contents of each file.
// The "depth" parameter is the current nesting level
// (for indentation).

static void
listRecursive(const DirectoryPrx& dir, int depth = 0)
{
    string indent(++depth, '\t');

    NodeSeq contents = dir->list();

    for(NodeSeq::const_iterator i = contents.begin();
i != contents.end(); ++i)
    {
        DirectoryPrx dir = DirectoryPrx::checkedCast(*i);
        FilePrx file = FilePrx::uncheckedCast(*i);
        cout << indent << (*i)->name()
<< (dir ? " (directory):" : " (file):") << endl;
        if(dir)
        {
            listRecursive(dir, depth);
        }
        else
        {
            Lines text = file->read();
            for(Lines::const_iterator j = text.begin(); j != text.end();
++j)
            {
                cout << indent << "\t" << *j << endl;
            }
        }
    }
}
```

The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the list operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `No de` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the `Node` *i s-a* `File` and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `chec kedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file) "` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned

sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Example of a File System Server in C++98
- Object Life Cycle
- IceGrid

Server-Side Slice-to-C++98 Mapping

The mapping for Slice data types to C++98 is identical on the client side and server side. This means that everything in Client-Side Slice-to-C++98 Mapping also applies to the server side. However, for the server side, there are a few additional things you need to know — specifically how to:

- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time.

Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Although the examples we present are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any C++98 mapping rules beyond those we describe here.

Topics

Server-Side C++98 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing virtual functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in C++

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The Slice compiler generates the following definition for this interface:

C++

```
namespace Filesystem
{
    class Node : public virtual Ice::Object
    {
    public:
        virtual std::string name(const Ice::Current& = Ice::emptyCurren
t) = 0;
        // ...
    };
    // ...
}
```

For the moment, we will ignore a number of other member functions of this class. The important points to note are:

- As for the client side, Slice modules are mapped to C++ namespaces with the same name, so the skeleton class definition is nested in the namespace `Filesystem`.
- The name of the skeleton class is the same as the name of the Slice interface (`Node`).
- The skeleton class contains a pure virtual member function for each operation in the Slice interface.
- The skeleton class is an abstract base class because its member functions are pure virtual.
- The skeleton class inherits from `Ice::Object` (which forms the root of the Ice object hierarchy).

Servant Classes in C++

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

C++

```
#include <Filesystem.h> // Slice-generated header

class NodeI : public virtual Filesystem::Node
{
public:
    NodeI(const std::string&);
    virtual std::string name(const Ice::Current&);
private:
    std::string _name;
};
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.)

Note that `NodeI` inherits from `Filesystem::Node`, that is, it derives from its skeleton class. It is a good idea to always use virtual inheritance when defining servant classes. Strictly speaking, virtual inheritance is necessary only for servants that implement interfaces that use multiple inheritance; however, the `virtual` keyword does no harm and, if you add multiple inheritance to an interface hierarchy half-way through development, you do not have to go back and add a `virtual` keyword to all your servant classes.

As far as Ice is concerned, the `NodeI` class must implement only a single member function: the pure virtual `name` function that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. Obviously, the constructor initializes the `_name` member and the `name` function returns its value:

C++

```
NodeI::NodeI(const std::string& name) : _name(name)
{
}

std::string
NodeI::name(const Ice::Current&) const
{
    return _name;
}
```

Normal and `idempotent` Operations in C++

The `name` member function of the `NodeI` skeleton is not a `const` member function. However, given that the operation does not modify the state of its object, it really should be a `const` member function. We can achieve this by adding the `["cpp:const"]` metadata directive. For example:

Slice

```
interface Example
{
                void normalOp();

    idempotent  void idempotentOp();

    ["cpp:const"]
    idempotent  void readonlyOp();
}
```

The skeleton class for this interface looks like this:

C++

```
class Example : public virtual Ice::Object
{
public:
    virtual void normalOp(const Ice::Current& = Ice::emptyCurrent) = 0;
    virtual void idempotentOp(const Ice::Current&
= Ice::emptyCurrent) = 0;
    virtual void readonlyOp(const Ice::Current&
= Ice::emptyCurrent) const = 0;
    // ...
};
```

Note that `readonlyOp` is mapped as a `const` member function due to the `["cpp:const"]` metadata directive; normal and `idempotent` operations (without the metadata directive) are mapped as ordinary, non-`const` member functions.

See Also

- Slice for a Simple File System
- C++98 Mapping for Interfaces
- Parameter Passing in C++98
- Raising Exceptions in C++98

Parameter Passing in C++98

Parameter passing on the server side generally follows the same rules as for the client side. Additionally, every operation receives a trailing parameter of type `Ice::Current`. For example, the `name` operation of the `Node` interface has no parameters, but the corresponding `name` method of the servant interface has a single parameter of type `Current`. We will ignore this parameter for now.

On this page:

Server-Side Mapping for Parameters in C++98

For each parameter of a Slice operation, the C++ mapping generates a corresponding parameter for the virtual member function in the skeleton. Parameter passing on the server side follows these rules:

- in-parameters are passed by value or `const` reference depending on the parameter type
- out-parameters are passed by reference
- return values are passed by value
- optional parameters are enclosed in `IceUtil::Optional` values

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
module M
{
    interface Example
    {
        string op(string sin, out string sout);
    }
}
```

The generated skeleton class for this interface looks as follows:

C++

```
namespace M
{
    class Example : public virtual Ice::Object
    {
    public:
        virtual std::string op(const std::string&, std::string&,

const Ice::Current& = Ice::emptyCurrent) = 0;
        // ...
    };
}
```

As you can see, there are no surprises here. For example, we could implement op as follows:

C++

```
std::string
ExampleI::op(const std::string& sin, std::string& sout,
const Ice::Current&)
{
    cout << sin << endl;        // In parameters are initialized
    sout = "Hello World!";      // Assign out parameter
    return "Done";              // Return a string
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal C++ rules and do not require special-purpose API calls or memory management.

Thread-Safe Marshaling in C++

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For C++ applications, this can affect servant methods that return instances of Slice classes or types referencing Slice classes.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following Slice:

Slice

```
sequence<int> IntSeq;
sequence<IntSeq> IntIntSeq;
sequence<string> StringSeq;
class Grid
{
    StringSeq xLabels;
    StringSeq yLabels;
    IntIntSeq values;
}

interface GridIntf
{
   Grid getGrid();
   void clearValues();
}
```

And the following servant implementation:

C++

```
class GridIntfI : public GridIntf
{
public:
    GridPtr getGrid(const Ice::Current&);
    void clear(const Ice::Current&);

private:
    IceUtil::Mutex _mutex;
    GridPtr _grid;
};

GridPtr
GridIntfI::getGrid(const Ice::Current&)
{
    IceUtil::Mutex::Lock lock(_mutex);
    return _grid;
}

void
GridIntfI::clearValues(const Ice::Current&)
{
    IceUtil::Mutex::Lock lock(_mutex);
    _grid->values.clear();
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned class in preparation to send a reply message, it is possible for another thread to dispatch the `clearValues` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `clearValues` operations does not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `clearValues` replaces `_grid` with a copy that contains empty values, leaving the previous contents of `_grid` unchanged:

C++

```
void GridIntfI::clearValues(const Ice::Current&)
{
    IceUtil::Mutex::Lock lock(_mutex);
    GridPtr grid = new Grid;
    grid->xLabels = _grid->xLabels;
    grid->yLabels = _grid->yLabels;
    _grid = grid;
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the `values` array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `amd` metadata directive changes the signature of the corresponding servant method to use asynchronous dispatch. After applying the metadata to the `getGrid` operation, we can revise the implementation as follows:

C++

```
class GridIntfI : public GridIntf
{
public:
    void getGrid_async(const AMD_GridIntf_getGridPtr&, const
Ice::Current&);
    void clear(const Ice::Current&);

private:
    IceUtil::Mutex _mutex;
    GridPtr _grid;
};

void
GridIntfI::getGrid_async(const AMD_GridIntf_getGridPtr& cb, const
Ice::Current&)
{
    IceUtil::Mutex::Lock lock(_mutex);
    cb->ice_response(_grid);
}
```

Normally, AMD is used in situations where the servant needs to delay its response to the client without blocking the calling thread. For `getGrid`, that is not the goal; instead, as a side-effect, AMD provides the desired marshaling behavior. Specifically, the Ice run time marshals the reply to an asynchronous request at the time the servant invokes `ice_response` on the AMD callback object. Because `getGrid` and `clearValues` are synchronized, this guarantees that the data remains in a consistent state during marshaling.

> The C++11 mapping offers another solution for marshaling immediately that does not require AMD.

See Also

- Server-Side C++98 Mapping for Interfaces
- C++98 Mapping for Operations
- C++98 Mapping for Optional Values
- Raising Exceptions in C++98
- The Current Object

Raising Exceptions in C++98

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

C++

```
void
Filesystem::FileI::write(const Filesystem::Lines& text,
const Ice::Current&)
{
    // Try to write the file contents here...
    // Assume we are out of space...
    if(error)
    {
        Filesystem::GenericError e;
        e.reason = "file too large";
        throw e;
    }
}
```

No memory management issues arise in the presence of exceptions.

Note that the Slice compiler never generates exception specifications for operations, regardless of whether the corresponding Slice operation definition has an exception specification or not. This is deliberate: C++ exception specifications do not add any value and are therefore not used by the Ice C++ mapping [1].

If you throw an arbitrary C++ exception (such as an `int` or other unexpected type), the Ice run time catches the exception and then returns an `UnknownException` to the client.

The server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw a run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

See Also

- Run-Time Exceptions
- C++98 Mapping for Exceptions
- Server-Side C++98 Mapping for Interfaces
- Parameter Passing in C++98

References

1. Sutter, H. 2002. A Pragmatic Look at Exception Specifications. C/C++ Users Journal 20 (7): 59-64.

Object Incarnation in C++98

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must follow the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a C++ Servant

Instantiating a servant means to allocate an instance on the heap:

C++

```
NodePtr servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance on the heap and assigns its address to a smart pointer of type `NodePtr`. This works because `NodeI` is derived from `Node`, so a smart pointer of type `NodePtr` can also look after an instance of type `NodeI`. However, if we want to invoke a member function of the derived `NodeI` class at this point, we have a problem: we cannot access member functions of the derived `NodeI` class through a `NodePtr` smart pointer, only member functions of `Node` base class. (The C++ type rules prevent us from accessing a member of a derived class through a pointer to a base class.) To get around this, we can modify the code as follows:

C++

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI("Fred");
```

This code makes use of the smart pointer template by defining `NodeIPtr` as a smart pointer to `NodeI` instances. Whether you use a smart pointer of type `NodePtr` or `NodeIPtr` depends solely on whether you want to invoke a member function of the `NodeI` derived class; if you only want to invoke member functions that are defined in the `Node` skeleton base class, it is sufficient to use a `NodePtr` and you need not define the `NodeIPtr` type.

Whether you use `NodePtr` or `NodeIPtr`, the advantages of using a smart pointer class should be obvious from the smart pointer discussion: they make it impossible to accidentally leak memory.

Creating an Identity in C++

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

C++

```
Ice::Identity id;
id.name = "Fred"; // Not unique, but good enough for now
```

Activating a C++ Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

C++

```
_adapter->add(servant, id);
```

Note the two arguments to `add`: the smart pointer to the servant and the object identity. Calling `add` on the object adapter adds the servant pointer and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant pointer from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

Servant Life Time and Reference Counts

Putting the preceding points together, we can write a simple function that instantiates and activates one of our `NodeI` servants. For this example, we use a simple helper function called `activateServant` that creates and activates a servant with a given identity:

C++

```
void
activateServant(const string& name)
{
    NodePtr servant = new NodeI(name);          // Refcount == 1
    Ice::Identity id;
    id.name = name;
    _adapter->add(servant, id);                 // Refcount == 2
}                                               // Refcount == 1
```

Note that we create the servant on the heap and that, once `activateServant` returns, we lose the last remaining handle to the servant (because the `servant` variable goes out of scope). The question is, what happens to the heap-allocated servant instance? The answer lies in the smart pointer semantics:

- When the new servant is instantiated, its reference count is initialized to 0.
- Assigning the servant's address to the `servant` smart pointer increments the servant's reference count to 1.
- Calling `add` passes the `servant` smart pointer to the object adapter which keeps a copy of the handle internally. This increments the reference count of the servant to 2.
- When `activateServant` returns, the destructor of the `servant` variable decrements the reference count of the servant to 1.

The net effect is that the servant is retained on the heap with a reference count of 1 for as long as the servant is in the servant map of its object adapter. (If we deactivate the servant, that is, remove it from the servant map, the reference count drops to zero and the memory occupied by the servant is reclaimed; we discuss these life cycle issues in Object Life Cycle.)

UUIDs as Identities in C++

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. The `Ice` namespace contains a helper function to create such identities:

C++

```
#include <Ice/Ice.h>
#include <iostream>

using namespace std;

int
main()
{
    cout << Ice::generateUUID() << endl;
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can rewrite the code shown earlier like this:

C++

```
void
activateServant(const string& name)
{
    NodePtr servant = new NodeI(name);
    _adapter->addWithUUID(servant);
}
```

Creating Proxies in C++

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Hello World Application. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in C++

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

C++

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI(name);
NodePrx proxy = NodePrx::uncheckedCast(_adapter->addWithUUID(servant));

// Pass proxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice::ObjectPrx`.

Direct Proxy Creation in C++

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

C++

```
Ice::Identity id;
id.name = Ice::generateUUID();
ObjectPrx o = _adapter->createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Hello World Application
- Object Adapter States
- Servant Locators
- Object Life Cycle

Asynchronous Method Dispatch (AMD) in C++98

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool. If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run time.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

On this page:

Enabling AMD with Metadata in C++

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

Slice

```
["amd"] interface I
{
    bool isValid();
    float computeRate();
}

interface J
{
    ["amd"] void startProcess();
    int endProcess();
}
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface or class level) minimizes the amount of generated code and, more importantly, minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

AMD Mapping in C++

The C++ mapping emits the following code for each AMD operation:

1. A callback class used by the implementation to notify the Ice run time about the completion of an operation. The name of this class is formed using the pattern `AMD_`*`class_op`*. For example, an operation named `foo` defined in interface `I` results in a class named `AMD_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Several methods are provided:
   - `void ice_response(<params>);`
     The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-`void` return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's `out` parameters follow the return value, in the order of declaration.
   - `void ice_exception(const std::exception &);`
     This version of `ice_exception` allows the server to raise any standard C++ exception, Ice run-time exception, or Ice user exception.
   - `void ice_exception();`
     This version of `ice_exception` allows the server to report an `UnknownException`.

   Neither `ice_response` nor `ice_exception` throw any exceptions to the caller.
2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the in-parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

Slice

```
interface I
{
    ["amd"] int foo(short s, out long l);
}
```

The callback class generated for operation `foo` is shown below:

C++

```
class AMD_I_foo : public ...
{
public:
    void ice_response(Ice::Int, Ice::Long);
    void ice_exception(const std::exception&);
    void ice_exception();
};
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

C++

```
void foo_async(const AMD_I_fooPtr&, Ice::Short);
```

AMD Exceptions in C++

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).

These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run-time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated and local exceptions may undergo translation.

AMD Example in C++

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

Slice

```
module Demo
{
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {}

    interface Model
    {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    }
}
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo::Model` and supplies a definition for the `interpolate_async` method:

C++

```
class ModelI : public virtual Demo::Model, public
virtual IceUtil::Mutex
{
public:
    virtual void interpolate_async(
        const Demo::AMD_Model_interpolatePtr&,
        const Demo::Grid&,
        Ice::Float,
        const Ice::Current&);

private:
    std::list<JobPtr> _jobs;
};
```

The implementation of `interpolate_async` uses synchronization to safely record the callback object and arguments in a `Job` that is added to a queue:

C++

```
void ModelI::interpolate_async(
    const Demo::AMD_Model_interpolatePtr& cb,
    const Demo::Grid& data,
    Ice::Float factor,
    const Ice::Current& current)
{
    IceUtil::Mutex::Lock sync(*this);
    JobPtr job = new Job(cb, data, factor);
    _jobs.push_back(job);
}
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute` to perform the interpolation. `Job` is defined as follows:

C++

```
class Job : public IceUtil::Shared
{
public:
    Job(const Demo::AMD_Model_interpolatePtr&, const Demo::Grid&,
Ice::Float);
    void execute();

private:
    bool interpolateGrid();

    Demo::AMD_Model_interpolatePtr _cb;
    Demo::Grid _grid;
    Ice::Float _factor;
};
typedef IceUtil::Handle<Job> JobPtr;
```

The implementation of `execute` uses `interpolateGrid` (not shown) to perform the computational work:

C++

```
Job::Job(const Demo::AMD_Model_interpolatePtr& cb,
const Demo::Grid& grid, Ice::Float factor) :
    _cb(cb), _grid(grid), _factor(factor)
{
}

void Job::execute()
{
    if(!interpolateGrid())
    {
        _cb->ice_exception(Demo::RangeError());
        return;
    }
    _cb->ice_response(_grid);
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

See Also

- Asynchronous Method Invocation (AMI) in C++98
- The Ice Threading Model
- User Exceptions
- Run-Time Exceptions

Example of a File System Server in C++98

This page presents the source code for a C++ server that implements our file system and communicates with the client we wrote earlier. The code is fully functional, apart from the required interlocking for threads.

The server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

On this page:

Implementing a File System Server in C++

We have now seen enough of the server-side C++ mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of two source files:

- `Server.cpp`
  This file contains the server main program.
- `FilesystemI.cpp`
  This file contains the implementation for the file system servants.

Server `main` Program in C++

Our server main program, in the file `Server.cpp`, consists of two functions, `main` and `run`. `main` creates and destroys an Ice communicator, and `run` uses this communicator instantiate our file system objects:

C++

```
#include <Ice/Ice.h>
#include <FilesystemI.h>

using namespace std;
using namespace Filesystem;

int run();

//
// Global variable for shutdownCommunicator
//
Ice::CommunicatorPtr communicator;

//
// Callback for CtrlCHandler
//
void
shutdownCommunicator(int)
{
```

```
    communicator->shutdown();
    cerr << "received signal, shutting down" << endl;
}

int
main(int argc, char* argv[])
{
    int status = 0;

    try
    {
        //
        // CtrlCHandler must be created before the communicator or any
other threads are started
        //
        Ice::CtrlCHandler ctrlCHandler;

        //
        // CommunicatorHolder's ctor initializes an Ice communicator,
        // and it's dtor destroys this communicator.
        //
        Ice::CommunicatorHolder ich(argc, argv);
        communicator = ich.communicator();

        //
        // Shutdown communicator on Ctrl-C
        //
        ctrlCHandler.setCallback(&shutdownCommunicator);

        //
        // The communicator initialization removes all Ice-related
arguments from argc/argv
        //
        if(argc > 1)
        {
            cerr << argv[0] << ": too many arguments" << endl;
            status = 1;
        }
        else
        {
            status = run();
        }
    }
    catch(const std::exception& ex)
    {
        cerr << ex.what() << endl;
        status = 1;
    }

    return status;
```

```
}

int
run()
{
    //
    // Create an object adapter.
    //
    Ice::ObjectAdapterPtr adapter =

communicator->createObjectAdapterWithEndpoints("SimpleFilesystem",
"default -h localhost -p 10000");

    //
    // Create the root directory (with name "/" and no parent)
    //
    DirectoryIPtr root = new DirectoryI("/", 0);
    root->activate(adapter);

    //
    // Create a file called "README" in the root directory
    //
    FileIPtr file = new FileI("README", root);
    Lines text;
    text.push_back("This file system contains a collection of poetry.");
    file->write(text);
    file->activate(adapter);

    //
    // Create a directory called "Coleridge" in the root directory
    //
    DirectoryIPtr coleridge = new DirectoryI("Coleridge", root);
    coleridge->activate(adapter);

    //
    // Create a file called "Kubla_Khan" in the Coleridge directory
    //
    file = new FileI("Kubla_Khan", coleridge);
    text.erase(text.begin(), text.end());
    text.push_back("In Xanadu did Kubla Khan");
    text.push_back("A stately pleasure-dome decree:");
    text.push_back("Where Alph, the sacred river, ran");
    text.push_back("Through caverns measureless to man");
    text.push_back("Down to a sunless sea.");
    file->write(text);
    file->activate(adapter);

    //
    // All objects are created, allow client requests now
    //
```

```
    adapter->activate();

    //
    // Wait until we are done
    //
```

```
    communicator->waitForShutdown();
    return 0;
}
```

There is quite a bit of code here, so let us examine each section in detail:

C++

```
#include <Ice/Ice.h>
#include <FilesystemI.h>

using namespace std;
using namespace Filesystem;
```

The code includes the header file `FilesystemI.h`. That file includes the header file that is generated by the Slice compiler, `Filesystem.h`.

Two `using` declarations, for the namespaces `std` and `Filesystem`, permit us to be a little less verbose in the source code.

The next part of the source code is the `main` function:

C++

```
try
    {
        //
        // CtrlCHandler must be created before the communicator or any
other threads are started
        //
        Ice::CtrlCHandler ctrlCHandler;

        //
        // CommunicatorHolder's ctor initializes an Ice communicator,
        // and it's dtor destroys this communicator.
        //
        Ice::CommunicatorHolder ich(argc, argv);
        communicator = ich.communicator();

        //
        // Shutdown communicator on Ctrl-C
        //
        ctrlCHandler.setCallback(&shutdownCommunicator);
        ...
```

We create a CtrlCHandler object which allows us to catch CTRL-C and similar signals in a portable fashion in C++. When the server receives such a signal, it shuts down the communicator. Shutting down the communicator in turn makes `waitForShutdown` return.

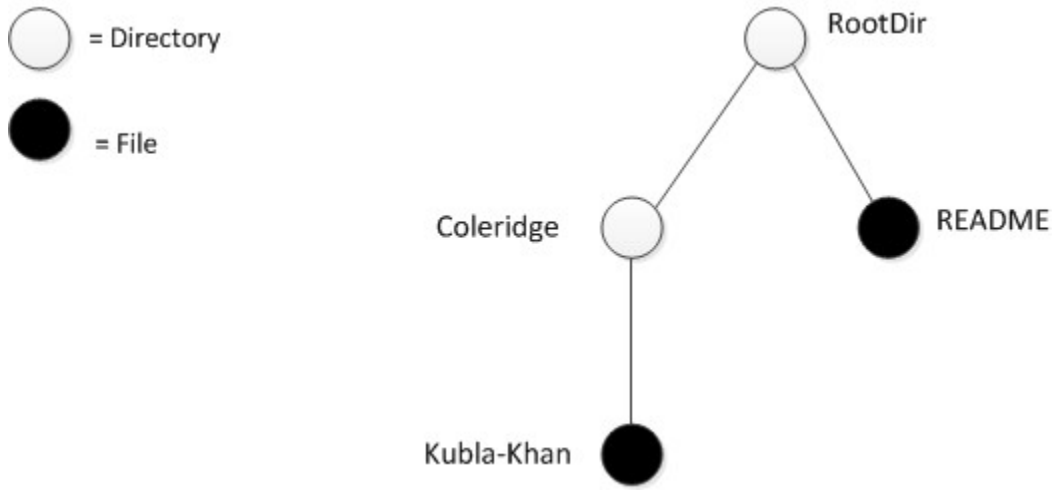
`main` later calls `run`, which is typically blocked on `waitForShutdown`:

C++

```
int
run()
{
    //
    // Create an object adapter.
    //
    Ice::ObjectAdapterPtr adapter =
communicator->createObjectAdapterWithEndpoints(
        "SimpleFilesystem", "default -h localhost -p 10000");

    ...


    //
    // All objects are created, allow client requests now
    //
    adapter->activate();

    //
    // Wait until we are done
    //
    communicator->waitForShutdown();

    return 0;
}
```

Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts three parameters: the communicator, the name of the directory or file to be created, and a handle to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent handle.) Thus, the statement

C++

```
DirectoryIPtr root = new DirectoryI(communicator(), "/", 0);
```

creates the root directory, with the name `"/"` and no parent directory. Note that we use the smart pointer class to hold the return value from `new`; that way, we avoid any memory management issues. The types `DirectoryIPtr` and `FileIPtr` are defined as follows in a header file `FilesystemI.h`:

C++

```
typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;
typedef IceUtil::Handle<FileI> FileIPtr;
```

Here is the code that establishes the structure in the illustration above.

C++

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI(communicator(), "/", 0);
        root->activate(adapter);

        // Create a file called "README" in the root directory
        //
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains a collection of poetr
y.");
        file->write(text);
        file->activate(adapter);

        // Create a directory called "Coleridge"
        // in the root directory
        //
        DirectoryIPtr coleridge =
new DirectoryI(communicator(), "Coleridge", root);
        coleridge->activate(adapter);

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = new FileI(communicator(), "Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);
        file->activate(adapter);
```

We first create the root directory and a file `README` within the root directory. (Note that we pass the handle to the root directory as the parent pointer when we create the new node of type `FileI`.)

After creating each servant, the code calls `activate` on the servant. (We will see the definition of this member function shortly.) The `activate` member function adds the servant to the ASM.

The next step is to fill the file with text:

C++

```
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains a collection of poetr
y.");
        file->write(text);
        file->activate(adapter);
```

Recall that Slice sequences map to STL vectors. The Slice type `Lines` is a sequence of strings, so the C++ type `Lines` is a vector of strings; we add a line of text to our `README` file by calling `push_back` on that vector.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

C++

```
        file->write(text);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a smart class pointer (of type `FilePtr`) and not via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary C++ function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the above illustration.

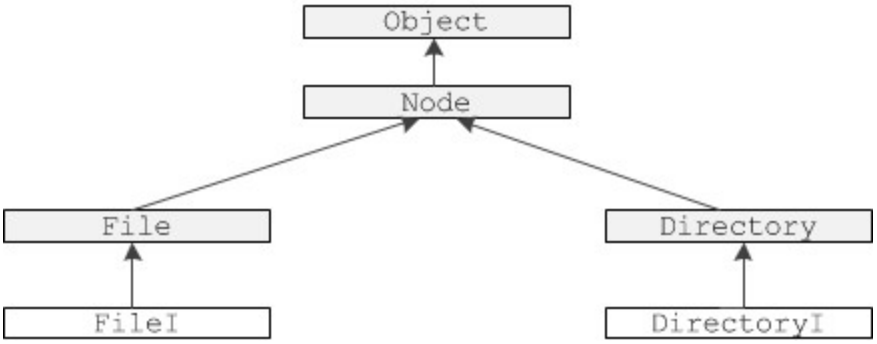
Servant Class Definitions in C++

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the C++ classes `FileI` and `DirectoryI`. This means that our servant classes might look as follows:

C++

```
namespace Filesystem
{
    class FileI : public virtual File
    {
      // ...
    };

    class DirectoryI : public virtual Directory
    {
       // ...
    };
}
```

This leads to the C++ class structure as shown:



File system servants using interface inheritance.

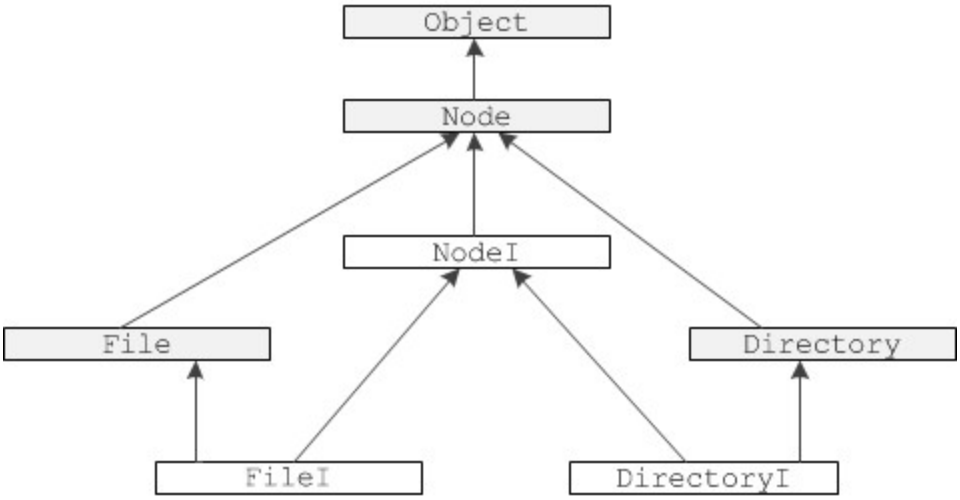
The shaded classes in the illustration above are skeleton classes and the unshaded classes are our servant implementations. If we implement our servants like this, `FileI` must implement the pure virtual operations it inherits from the `File` skeleton (`read` and `write`), as well as the operation it inherits from the `Node` skeleton (`name`). Similarly, `DirectoryI` must implement the pure virtual function it inherits from the `Directory` skeleton (`list`), as well as the operation it inherits from the `Node` skeleton (`name`). Implementing the servants in this way uses interface inheritance from `Node` because no implementation code is inherited from that class.

Alternatively, we can implement our servants using the following definitions:

C++

```
namespace Filesystem
{
    class NodeI : public virtual Node
    {
      // ...
    };

    class FileI : public virtual File, public virtual NodeI
    {
      // ...
    };

    class DirectoryI : public virtual Directory, public virtual NodeI
    {
      // ...
    };
}
```

This leads to the C++ class structure shown:



File system servants using implementation inheritance.

In this implementation, `NodeI` is a concrete base class that implements the `name` operation it inherits from the `Node` skeleton. `FileI` and `DirectoryI` use multiple inheritance from `NodeI` and their respective skeletons, that is, `FileI` and `DirectoryI` use implementation inheritance from their `NodeI` base class.

Either implementation approach is equally valid. Which one to choose simply depends on whether we want to re-use common code provided by `NodeI`. For the implementation that follows, we have chosen the second approach, using implementation inheritance.

Given the structure in the above illustration and the operations we have defined in the Slice definition for our file system, we can add these operations to the class definition for our servants:

C++

```
namespace Filesystem
{
    class NodeI : public virtual Node
    {
    public:
        virtual std::string name(const Ice::Current&);
    };

    class FileI : public virtual File, public virtual NodeI
    {
    public:
        virtual Lines read(const Ice::Current&);
        virtual void write(const Lines&, const Ice::Current&);
    };

    class DirectoryI : public virtual Directory, public virtual NodeI
    {
    public:
        virtual NodeSeq list(const Ice::Current&);
    };
}
```

This simply adds signatures for the operation implementations to each class. Note that the signatures must exactly match the operation signatures in the generated skeleton classes — if they do not match exactly, you end up overloading the pure virtual function in the base

class instead of overriding it, meaning that the servant class cannot be instantiated because it will still be abstract. To avoid signature mismatches, you can copy the signatures from the generated header file (`Filesystem.h`), or you can use the `--impl` option with `slice2cpp` to generate header and implementation files that you can add your application code to.

Now that we have the basic structure in place, we need to think about other methods and data members we need to support our servant implementation. Typically, each servant class hides the copy constructor and assignment operator, and has a constructor to provide initial state for its data members. Given that all nodes in our file system have both a name and a parent directory, this suggests that the `NodeI` class should implement the functionality relating to tracking the name of each node, as well as the parent-child relationships:

C++

```
namespace Filesystem
{
    class DirectoryI;
    typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

    class NodeI : public virtual Node
    {
    public:
        virtual std::string name(const Ice::Current&);
        NodeI(const Ice::CommunicatorPtr&, const std::string&,
const DirectoryIPtr&);
        void activate(const Ice::ObjectAdapterPtr&);
    private:
        std::string _name;
        Ice::Identity _id;
        DirectoryIPtr _parent;
        NodeI(const NodeI&);               // Copy forbidden
        void operator=(const NodeI&);      // Assignment forbidden
    };
}
```

The `NodeI` class has a private data member to store its name (of type `std::string`) and its parent directory (of type `DirectoryIPtr`). The constructor accepts parameters that set the value of these data members. For the root directory, by convention, we pass a null handle to the constructor to indicate that the root directory has no parent. The constructor also requires the communicator to be passed to it. This is necessary because the constructor creates the identity for the servant, which requires access to the communicator. The `activate` member function adds the servant to the ASM (which requires access to the object adapter) and connects the child to its parent.

The `FileI` servant class must store the contents of its file, so it requires a data member for this. We can conveniently use the generated `Lines` type (which is a `std::vector<std::string>`) to hold the file contents, one string for each line. Because `FileI` inherits from `NodeI`, it also requires a constructor that accepts the communicator, file name, and parent directory, leading to the following class definition:

C++

```
namespace Filesystem
{
    class FileI : public virtual File, public virtual NodeI
    {
    public:
        virtual Lines read(const Ice::Current&);
        virtual void write(const Lines&, const Ice::Current&);
        FileI(const Ice::CommunicatorPtr&, const std::string&,
const DirectoryIPtr&);
    private:
        Lines _lines;
    };
}
```

For directories, each directory must store its list of child notes. We can conveniently use the generated `NodeSeq` type (which is a `vector<NodePrx>`) to do this. Because `DirectoryI` inherits from `NodeI`, we need to add a constructor to initialize the directory name and its parent directory. As we will see shortly, we also need a private helper function, `addChild`, to make it easier to connect a newly created directory to its parent. This leads to the following class definition:

C++

```
namespace Filesystem
{
    class DirectoryI : public virtual Directory, public virtual NodeI
    {
    public:
        virtual NodeSeq list(const Ice::Current&) const;
        DirectoryI(const Ice::CommunicatorPtr&, const std::string&,
const DirectoryIPtr&);
        void addChild(NodePrx child);
    private:
        NodeSeq _contents;
    };
}
```

Servant Header File Example

Putting all this together, we end up with a servant header file, `FilesystemI.h`, as follows:

C++

```
#include <Ice/Ice.h>
#include <Filesystem.h>

namespace Filesystem
{
    class DirectoryI;
    typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

    class NodeI : public virtual Node
    {
    public:
        virtual std::string name(const Ice::Current&);
        NodeI(const Ice::CommunicatorPtr&, const std::string&,
const DirectoryIPtr&);
        void activate(const Ice::ObjectAdapterPtr&);
    private:
        std::string _name;
        Ice::Identity _id;
        DirectoryIPtr _parent;
        NodeI(const NodeI&);            // Copy forbidden
        void operator=(const NodeI&);   // Assignment forbidden
    };

    typedef IceUtil::Handle<NodeI> NodeIPtr;

    class FileI : public virtual File, public virtual NodeI
    {
    public:
        virtual Lines read(const Ice::Current&);
        virtual void write(const Lines&,
const Ice::Current& = Ice::emptyCurrent);
        FileI(const Ice::CommunicatorPtr&, const std::string&,
const DirectoryIPtr&);
    private:
        Lines _lines;
    };

    typedef IceUtil::Handle<FileI> FileIPtr;

    class DirectoryI : public virtual Directory, public virtual NodeI
    {
    public:
        virtual NodeSeq list(const Ice::Current&);
        DirectoryI(const Ice::CommunicatorPtr&, const std::string&,
const DirectoryIPtr&);
        void addChild(const Filesystem::NodePrx&);
    private:
        Filesystem::NodeSeq _contents;
    };
}
```

The Servant Implementation in C++

The implementation of our servants is mostly trivial, following from the class definitions in our `FilesystemI.h` header file.

Implementing `FileI`

The implementation of the `read` and `write` operations for files is trivial: we simply store the passed file contents in the `_lines` data member. The constructor is equally trivial, simply passing its arguments through to the `NodeI` base class constructor:

C++

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
const Ice::Current&)
{
    _lines = text;
}

Filesystem::FileI::FileI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}
```

Implementing `DirectoryI`

The implementation of `DirectoryI` is equally trivial: the `list` operation simply returns the `_contents` data member and the constructor passes its arguments through to the `NodeI` base class constructor:

C++

```
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current&)
{
    return _contents;
}

Filesystem::DirectoryI::DirectoryI(const
Ice::CommunicatorPtr& communicator,
                                   const string& name,
                                   const DirectoryIPtr& parent)
    : NodeI(name, parent)
{
}

void
Filesystem::DirectoryI::addChild(const NodePrx child)
{
    _contents.push_back(child);
}
```

The only noteworthy thing is the implementation of `addChild`: when a new directory or file is created, the constructor of the `NodeI` base class calls `addChild` on its own parent, passing it the proxy to the newly-created child. The implementation of `addChild` appends the passed reference to the contents list of the directory it is invoked on (which is the parent directory).

Implementing `NodeI`

The name operation of our `NodeI` class is again trivial: it simply returns the `_name` data member:

C++

```
std::string
Filesystem::NodeI::name(const Ice::Current&)
{
    return _name;
}
```

The `NodeI` constructor creates an identity for the servant:

C++

```
Filesystem::NodeI::NodeI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : _name(name), _parent(parent)
{
    _id.name = parent ? IceUtil::generateUUID() : "RootDir";
}
```

For the root directory, we use the fixed identity `"RootDir"`. This allows the client to create a proxy for the root directory. For directories other than the root directory, we use a UUID as the identity.

Finally, `NodeI` provides the `activate` member function that adds the servant to the ASM and connects the child node to its parent directory:

C++

```
void
Filesystem::NodeI::activate(const Ice::ObjectAdapterPtr& a)
{
    NodePrx thisNode = NodePrx::uncheckedCast(a->add(this, _id));
    if(_parent)
    {
        _parent->addChild(thisNode);
    }
}
```

This completes our servant implementation. The complete source code is shown here once more:

C++

```
#include <IceUtil/IceUtil.h>
#include <FilesystemI.h>

using namespace std;

// Slice Node::name() operation

std::string
Filesystem::NodeI::name(const Ice::Current&)
{
    return _name;
}

// NodeI constructor

Filesystem::NodeI::NodeI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
```

```
                          const DirectoryIPtr& parent)
    : _name(name), _parent(parent)
{
    // Create an identity. The root directory has the fixed identity "R
ootDir"
    //
    _id.name = parent ? IceUtil::generateUUID() : "RootDir";
}

// NodeI activate() member function

void
Filesystem::NodeI::activate(const Ice::ObjectAdapterPtr& a)
{
    NodePrx thisNode = NodePrx::uncheckedCast(a->add(this, _id));
    if(_parent)
    {
        _parent->addChild(thisNode);
    }
}

// Slice File::read() operation

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

// Slice File::write() operation

void
Filesystem::FileI::write(const Filesystem::Lines& text, const Ice::Curr
ent&)
{
    _lines = text;
}

// FileI constructor

Filesystem::FileI::FileI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}

// Slice Directory::list() operation

Filesystem::NodeSeq
```

```
Filesystem::DirectoryI::list(const Ice::Current& c)
{
    return _contents;
}

// DirectoryI constructor

Filesystem::DirectoryI::DirectoryI(const Ice::CommunicatorPtr& communic
ator,
                                   const string& name,
                                   const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}

// addChild is called by the child in order to add
// itself to the _contents member of the parent

void
Filesystem::DirectoryI::addChild(const NodePrx& child)
{
```

```
    _contents.push_back(child);
}
```

See Also

- Slice for a Simple File System
- Example of a File System Client in C++98
- C++98 Mapping for Sequences
- slice2cpp Command-Line Options (C++98)
- UUIDs as Identities in C++

Slice-to-C++98 Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to C++98 is identical to the mapping for these constructs without the `local` qualifier. The generated C++ code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

The rest of this section describes the mapping of the remaining local types to C++98:

- C++98 Mapping for Local Interfaces
- C++98 Mapping for Local Classes
- C++98 Mapping for Local Exceptions
- C++98 Mapping for Operations on Local Types
- C++98 Mapping for Data Members in Local Types

C++98 Mapping for Local Interfaces

On this page:

Mapped C++ Class

A Slice local interface is mapped to a C++ abstract base class with the same name, for example:

Slice

```
module Ice
{
    local interface Communicator
    {
        ...
    }
}
```

is mapped to the C++ class `Communicator`:

C++

```
namespace Ice
{
    class Communicator : public virtual Ice::LocalObject
    {
        ...
    };
}
```

LocalObject in C++

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to class `Ice::LocalObject` in C++98:

C++

```
namespace Ice
{
    class LocalObject : public virtual IceUtil::Shared
    {
    public:
        virtual bool operator==(const LocalObject&) const;
        virtual bool operator<(const LocalObject&) const;
    };

    typedef IceInternal::Handle<LocalObject> LocalObjectPtr;
}
```

Mapping for Local Interface Inheritance in C++

Inheritance of local Slice interfaces is mapped to public virtual inheritance in C++. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

C++

```
namespace M
{
    class A : public virtual Ice::LocalObject { ... };
    class B : public virtual A { ... };
    class C : public virtual A { ... };
    class D : public virtual B, public virtual C { ... };
}
```

C++98 Mapping for Local Classes

On this page:

Mapped C++ Class

A local Slice class is mapped to a C++ class with the same name. For example:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ...
    }
}
```

is mapped to the C++ class `ConnectionInfo`:

C++

```
namespace Ice
{
    class ConnectionInfo : public virtual Ice::LocalObject
    {
        ...
    };
    typedef IceInternal::Handle<ConnectionInfo> ConnectionInfoPtr;
}
```

LocalObject in C++

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to `Ice::LocalObject` in C++98.

Mapping for Local Interface Inheritance in C++

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` for classes is mapped by default to simple inheritance in C++, while `implements` is mapped to public virtual inheritance. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

C++

```
namespace M
{
    class A : public virtual Ice::LocalObject { ... };
    class B : public virtual Ice::LocalObject { ... };

    class C : public virtual A, public virtual B { ... };
    class D : public C { ... };
}
```

C++98 Mapping for Local Exceptions

On this page:

Mapped C++ Class

A local Slice exception is mapped to a C++ class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the C++ class `InitializationException`:

C++

```
namespace Ice
{
    class InitializationException : public Ice::LocalException
    {
        ...
    };
}
```

Base Class for Local Exceptions in C++

All mapped C++ exception classes derive directly or indirectly from `Ice::LocalException`:

C++

```
namespace Ice
{
    class LocalException : public Exception
    {
    public:
        LocalException(const char*, int);
        virtual ~LocalException() throw();

        virtual LocalException* ice_clone() const;

        static const std::string& ice_staticId();
    };
}
```

All these member functions are described on the C++98 Mapping for Exceptions page.

Mapping for Local Exception Inheritance in C++

A local Slice exception can extend another Slice exception. In C++, this is mapped to simple public inheritance. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

C++

```
namespace M
{
    class ErrorBase : public Ice::LocalException { ... };
    class ResourceError : public ErrorBase { ... };
}
```

C++98 Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to a pure virtual member function with the same name. The mapping of operation parameters to C++ is identical to the Client-Side Mapping for these parameters:

- in parameters are passed by value (for simple types such as integers), or by const reference (for other types)
- out parameters are passed by reference
- return values are mapped to return values in C++

However, unlike the Client-Side mapping, there is no trailing `Ice::Context` parameter in the mapped member functions.

The type of a parameter can be a local interface or class. Such a parameter is passed as a *<mapped C++ class>* `Ptr`.

A `LocalObject` parameter is mapped to a parameter of type `Ice::LocalObjectPtr`. You can also create constructed types (such as local sequences and local dictionaries) with local types.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

C++

```
namespace M
{
    class L;
    typedef IceInternal::Handle<L> LPtr;
    typedef std::vector<LPtr> LSeq;
    class L : public virtual Ice::Object
    {
    public:
        virtual ~L();
        virtual std::string op(int n, const std::string&, const
Ice::LocalObjectPtr& any,
                               int& m, std::string& t, LSeq& newLSeq) =
0;
    };
}
```

C++98 Mapping for Data Members in Local Types

Data members on local Slice types (classes, exceptions and structs) are mapped to C++ just like the data members of the corresponding non local Slice construct.

A local Slice type can have a data member of type local interface or class, which is mapped to a C++ data member of type `<`*`mapped C++ class`*`>Ptr`. A `LocalObject` data member is mapped to a C++ data member of the same name with type `Ice::LocalObjectPtr`.

Customizing the C++98 Mapping

Ice for C++ allows you to map Slice strings, sequences and dictionaries to your own C++ types, through the `cpp:type` and `cpp:view-type` metadata directives.

When you map a Slice type to a C++ type from the standard library type, Ice is able to marshal and unmarshal this type without your help. For example, if you map a `sequence<string>` to a `std::list<std::wstring>`, Ice knows exactly what to do. However, if you select a class or template outside the standard C++ library, such as your own C++ template, you usually need to tell Ice how to marshal and unmarshal this type.

The following pages describe the set of classes and templates that Ice uses for marshaling and unmarshaling, and how to plug-in your own C++ types into this framework.

Topics

The C++98 Stream Helpers

Ice for C++ uses a set of C++ templates and specializations of these templates to marshal and unmarshal C++ types. A type that can be marshaled and/or unmarshaled using this framework is known as a *Streamable.*

On this page:

Marshaling and Unmarshaling C++ Objects

Ice for C++ marshals C++ objects into streams of bytes, and unmarshals these C++ objects from streams of bytes. Ice for C++ has an internal implementation for these streams, and a publicly available implementation, and both implementations have mostly the same API.

To marshal a C++ object into a stream, you (or the generated code) can simply call `write(obj)` on that stream. For basic types, such as `int` or `string`, there is a corresponding write function on the stream. For other types, `write` is an inline template function that delegates to a specialization of the `StreamHelper` template class:

C++

```
namespace Ice
{
    class OutputStream : ...
    {
    public:
    ...
        template<typename T> inline void write(const T& v)
        {
            StreamHelper<T, StreamableTraits<T>::helper>::write(this,
v);
        }
    };
}
```

Likewise, when unmarshaling a C++ object from a stream, you (or the generated code) can simply call `read(obj)` on that stream, and the reading of most types delegates to a specialization of this `StreamHelper` template class:

C++

```
namespace Ice
{
    class InputStream : ...
    {
    public:
    ...
        template<typename T> inline void read(T& v)
        {
            StreamHelper<T, StreamableTraits<T>::helper>::read(this, v);
       }
    };
}
```

Ice looks for the `StreamHelper` specializations in namespace Ice, so they must all be in namespace Ice.

Each `StreamHelper` specialization typically provides a static `write` function to write the C++ object into the steam, and a static `read` funct ion to read the C++ object from the stream. The stream object in these `StreamHelper` static functions is represented by a template parameter type, which allows a `StreamHelper` specialization to work with any stream implementation:

C++

```
// Typical StreamHelper specialization
namespace Ice
{
    template<>
    struct StreamHelper<SomeType, StreamHelperCategoryXXX>
    {
        template<class S> static inline void
        write(S* stream, const SomeType& v)
        {
            ... marshal v...
        }
        template<class S> static inline void
        read(S* stream, SomeType& v)
        {
           ... unmarshal v ...
        }
    };
}
```

The base `StreamHelper` template class is not defined, since there is no default way to read a C++ object from a stream or write this object into a stream:

C++

```
namespace Ice
{
    template<typename T, StreamHelperCategory st> struct StreamHelper;
}
```

For most types, Ice for C++ provides or generates the corresponding `StreamHelper` specialization (or partial specialization), for example all Slice sequences mapped to a C++ vector, C++ list or similar type use this specialization:

C++

```
namespace Ice
{
    template<typename T>
    struct StreamHelper<T, StreamHelperCategorySequence>
    {
        template<class S> static inline void
        write(S* stream, const T& v)
        {
            stream->writeSize(static_cast<Int>(v.size()));
            for(typename T::const_iterator p = v.begin(); p != v.end();
++p)
            {
                stream->write(*p);
            }
        }
        template<class S> static inline void
        read(S* stream, T& v)
        {
            Int sz =
stream->readAndCheckSeqSize(StreamableTraits<typename
T::value_type>::minWireSize);
            T(sz).swap(v);
            for(typename T::iterator p = v.begin(); p != v.end(); ++p)
            {
                stream->read(*p);
            }
        }
    };
}
```

StreamableTraits

The `StreamHelper` specializations use specializations of the `StremableTraits` template class to retrieve some characteristics (or traits) of the C++ type it operates on:

C++

```
namespace Ice
{
    typedef int StreamHelperCategory;
    const StreamHelperCategory StreamHelperCategoryUnknown = 0;
    const StreamHelperCategory StreamHelperCategoryBuiltin = 1;
    const StreamHelperCategory StreamHelperCategoryStruct = 2;
    const StreamHelperCategory StreamHelperCategoryStructClass = 3;
    const StreamHelperCategory StreamHelperCategoryEnum = 4;
    const StreamHelperCategory StreamHelperCategorySequence = 5;
    const StreamHelperCategory StreamHelperCategoryDictionary = 6;
    const StreamHelperCategory StreamHelperCategoryProxy = 7;
    const StreamHelperCategory StreamHelperCategoryClass = 8;
    const StreamHelperCategory StreamHelperCategoryUserException = 9;

    template<typename T, typename Enabler = void>
    struct StreamableTraits
    {
        static const StreamHelperCategory helper = ...;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
}
```

In particular, `StreamHelper` specializations are divided in various categories, to allow a single partial specialization of `StreamHelper` to handle several similar C++ types. For example, Ice for C++ provides a single `StreamHelper` partial specialization for all `enum` types, with category `StreamHelperCategoryEnum`:

C++

```
template<typename T>
struct StreamHelper<T, StreamHelperCategoryEnum>
{
    template<class S> static inline void
    write(S* stream, const T& v)
    {
        ...sanity check...
        stream->writeEnum(static_cast<Int>(v),
StreamableTraits<T>::maxValue);
    }
    template<class S> static inline void
    read(S* stream, T& v)
    {
        Int value = stream->readEnum(StreamableTraits<T>::maxValue);
        ..sanity check...
        v = static_cast<T>(value);
    }
};
```

`StreamHelperCategory` is an integer (`int`). Values between 0 and 20 are reserved for Ice; you can use other values for your own `StreamHelper` specializations.

A `StreamableTraits` is always defined in namespace Ice and provides three `static const` data members:

- `helper`
  The category of `StreamHelper` associated with this C++ type (see above). The base `StreamableTraits` template class computes `helper` automatically for maps, lists, vectors and similar types.

- `minWireSize`
  The minimum size (in bytes) of the corresponding Slice type when marshaled using the Ice Encoding. The base `StreamableTraits` template class provides the default value, 1 byte. This value should be one for sequences and dictionaries, since an empty sequence or dictionary is marshaled as a single byte (size = 0).

- `fixedLength`
  `true` when the corresponding Slice type is marshaled on a fixed number of bytes, and `false` when it is marshaled on a variable number of bytes. The default, provided by the base `StreamableTraits` template class, is false.

Ice provides `StreamableTraits` specializations for all the C++ types it uses when mapping Slice types to C++. For example, Ice provides a `StreamTraits<Ice::Long>` (for 64-bit signed integers) and a `StreamableTraits<std::string>`:

C++

```
namespace Ice
{
    template<>
    struct StreamableTraits<Long>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryBuiltin;
        static const int minWireSize = 8;
        static const bool fixedLength = true;
    };

    template<>
    struct StreamableTraits<std::string>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryBuiltin;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
}
```

For each constructed type, such as Slice `structs`, the Slice to C++ translator generates the corresponding `StreamableTraits` specialization. For example:

C++

```
// Generated C++ code
namespace Ice
{
    template<>
    struct StreamableTraits<::Ice::Identity>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryStruct;
        static const int minWireSize = 2;
        static const bool fixedLength = false;
    };
}
```

Marshaling and Unmarshaling Optional Values

As described in Data Encoding for Optional Values, the encoding or wire representation of an optional data member or parameter consists of:

- An optional format (a 3-bits value) that depends on the associated Slice type. For example, the optional format for a `sequence<string>` is `FSize` (or `6`).
- The tag associated with this optional data member or parameter (this tag is a positive integer)

- The actual value of this optional data member or parameter

To marshal an optional value into a stream, you (or the generated code) call `write(tag, obj)` on that stream, where `tag` is the optional's tag (an int), and obj is the optional value. `write` is a template function that uses a specialization of the `StreamOptionalHelper` template class when this optional value is set:

C++

```
namespace Ice
{
    class OutputStream : ...
    {
    public:
    ...
        template<typename T> inline void write(int tag, const
IceUtil::Optional<T>& v)
        {
            if(v)
            {
                writeOptional(tag, StreamOptionalHelper<T,

StreamableTraits<T>::helper,

StreamableTraits<T>::fixedLength>::optionalFormat);
                StreamOptionalHelper<T, StreamableTraits<T>::helper,
StreamableTraits<T>::fixedLength>::write(this, *v);
            }
        }
    };
}
```

Likewise, when unmarshaling an optional value from a stream, you (or the generated code) call `read(tag, obj)` on that stream, which delegates to a specialization of the same `StreamOptionalHelper` template class when the stream holds a value for this optional data member or parameter:

C++

```
namespace Ice
{
    class InputStream : ...
    {
    public:
    ...
        template<typename T> inline void read(int tag,
IceUtil::Optional<T>& v)
        {
            if(readOptional(tag, StreamOptionalHelper<T,

StreamableTraits<T>::helper,

StreamableTraits<T>::fixedLength>::optionalFormat))
            {
                v.__setIsSet();
                StreamOptionalHelper<T, StreamableTraits<T>::helper,
StreamableTraits<T>::fixedLength>::read(this, *v);
            }
            else
            {
                v = IceUtil::None;
            }
        }
    };
}
```

Each `StreamOptionalHelper` specialization usually provides `static read` and `write` functions, just like `StreamHelper` specializations, and also an `optionalFormat` static data member that provides the optional format of the corresponding Slice type.

For many types, the `StreamOptionalHelper` specialization simply delegates to this type's `StreamHelper` specialization, and computes `optionalFormat` with the `GetOptionalFormat` template:

C++

```
// GetOptionalFormat is declared but never defined
template<StreamHelperCategory st, int minWireSize, bool fixedLength>
struct GetOptionalFormat;

// Base StreamOptionalHelper template class
namespace Ice
{
    template<typename T, StreamHelperCategory st, bool fixedLength>
    struct StreamOptionalHelper
    {
        typedef StreamableTraits<T> Traits;

        static const OptionalFormat optionalFormat =
GetOptionalFormat<st, Traits::minWireSize, fixedLength>::value;

        template<class S> static inline void
        write(S* stream, const T& v)
        {
            stream->write(v);
        }
        template<class S> static inline void
        read(S* stream, T& v)
        {
            stream->read(v);
        }
    };
}
```

For example, the optional format for a long long int (64-bit signed integer, encoded on exactly 8 bytes) is provided by this specialization of `GetOptionalFormat`:

C++

```
namespace Ice
{
    template<>
    struct GetOptionalFormat<StreamHelperCategoryBuiltin, 8, true>
    {
        static const OptionalFormat value = OptionalFormatF8;
    };
}
```

See Also

- Data Encoding
- Dynamic Ice

The cpp:type and cpp:view-type Metadata Directives with C++98

Ice for C++ provides two metadata directives that allow you to map Slice types to arbitrary C++ types: `cpp:type` and `cpp:view-type`. The se metadata directives currently apply only to the following Slice types:

- `string`
- `sequence`
- `dictionary`

On this page:

Customizing the sequence mapping with cpp:type

The `cpp:type:`*`c++-type`* metadata directive allows you to map a given Slice type, data member or parameter to the C++ type of your choice.

For example, you can override the default mapping of a Slice sequence type:

Slice

```
[["cpp:include:list"]]

module Food
{
    enum Fruit { Apple, Pear, Orange }

    ["cpp:type:std::list<Food::Fruit>"]
    sequence<Fruit> FruitPlatter;
}
```

With this metadata directive, the Slice sequence now maps to a C++ `std::list` instead of the default `std::vector`:

C++

```
#include <list>

namespace Food
{
    typedef std::list<Food::Fruit> FruitPlatter;

    // ...
}
```

The Slice to C++ compiler takes the string following the `cpp:type:` prefix as the name of the mapped C++ type. For example, we could use `["cpp:type:::std::list< ::Food::Fruit>"]`. In that case, the compiler would use a fully-qualified name to define the type:

C++11

```
typedef ::std::list< ::Food::Fruit> FruitPlatter;
```

Note that the code generator inserts whatever string you specify following the `cpp:type:` prefix literally into the generated code. We recommend you use fully qualified names to avoid C++ compilation failures due to unknown symbols.

Also note that, to avoid compilation errors in the generated code, you must instruct the compiler to generate an appropriate include directive with the `cpp:include` global metadata directive. This causes the compiler to add the line

C++

```
#include <list>
```

to the generated header file.

In addition to modifying the type of a sequence itself, you can also modify the mapping for particular return values or parameters. For example:

Slice

```
[["cpp:include:list"]]
[["cpp:include:deque"]]

module Food
{
    enum Fruit { Apple, Pear, Orange }

    sequence<Fruit> FruitPlatter;

    interface Market
    {
        ["cpp:type:list< ::Food::Fruit>"]
        FruitPlatter
barter(["cpp:type:deque< ::Food::Fruit>"] FruitPlatter offer);
    }
}
```

With this definition, the default mapping of `FruitPlatter` to a C++ `vector` still applies but the return value of `barter` is mapped as a `list`, and the `offer` parameter is mapped as a `deque`.

Instead of `std::list` or `std::deque`, you can specify a type of your own as the sequence type, for example:

Slice

```
[["cpp:include:FruitBowl.h"]]

module Food
{
    enum Fruit { Apple, Pear, Orange }

    ["cpp:type:FruitBowl"]
    sequence<Fruit> FruitPlatter;
}
```

With these metadata directives, the compiler will use a C++ type `FruitBowl` as the sequence type, and add an `include` directive for the header file `FruitBowl.h` to the generated code.

The class or template class you provide must meet the following requirements:

- The class must have a default constructor.
- The class must have a copy constructor.

If you use a class that also meets the following requirements

- The class has a single-argument constructor that takes the size of the sequence as an argument of unsigned integral type.
- The class has a member function `size` that returns the number of elements in the sequence as an unsigned integral type.
- The class provides a member function `swap` that swaps the contents of the sequence with another sequence of the same type.
- The class defines `iterator` and `const_iterator` types and provides `begin` and `end` member functions with the usual semantics; its iterators are comparable for equality and inequality.

then you do not need to provide code to marshal and unmarshal your custom sequence – Ice will do it automatically.

Less formally, this means that if the provided class looks like a `vector`, `list`, or `deque` with respect to these points, you can use it as a custom sequence implementation without any additional coding.

Customizing the dictionary mapping with cpp:type

You can override the default mapping of Slice dictionaries to C++ maps with a `cpp:type` metadata directive, for example:

Slice

```
[["cpp:include:unordered_map"]]

["cpp:type:std::unordered_map<Ice::Long, Employee>"] dictionary<long,
Employee> EmployeeMap;
```

With this metadata directive, the dictionary now maps to a C++ `std::unordered_map`:

C++

```
#include <unordered_map>

typedef std::unordered_map<Ice::Long, Employee> EmployeeMap;
```

Like with sequences, anything following the `cpp:type:` prefix is taken to be the name of the type. For example, we could use `["cpp:type`

`:::std::unordered_map<int, std::string>"]`. In that case, the compiler would use a fully-qualified name to define the type:

C++11

```
typedef ::std::unordered_map<int, std::string> IntStringDict;
```

To avoid compilation errors in the generated code, you must instruct the compiler to generate an appropriate include directive with the `cpp:include` global metadata directive. This causes the compiler to add the line

C++

```
#include <unordered_map>
```

to the generated header file.

Instead of `std::unordered_map`, you can specify a type of your own as the dictionary type, for example:

Slice

```
[["cpp:include:CustomMap.h"]]

["cpp:type:MyCustomMap<Ice::Long, Employee>"] dictionary<long, Employee>
EmployeeMap;
```

With these metadata directives, the compiler will use a C++ type `MyCustomMap` as the dictionary type, and add an include directive for the header file `CustomMap.h` to the generated code.

The class or template class you provide must meet the following requirements:

- The class must have a default constructor.
- The class must have a copy constructor.
- The class must provide nested types named `key_type`, `mapped_type` and `value_type`.
- The class must provide `iterator` and `const_iterator` types and provide `begin` and `end` member functions with the usual semantics; these iterators must be comparable for equality and inequality.
- The class must provide a `clear` function.
- The class must provide an `insert` function that takes an `iterator` (as location hint) plus a `value_type` parameter, and returns an `iterator` to the new entry or to the existing entry with the given key.

Less formally, this means you can use any class or template class that looks like a standard `map` or `unordered_map` as your custom dictionary type.

In addition to modifying the type of a dictionary itself, you can also modify the mapping for particular return values or parameters. For example:

Slice

```
[["cpp:include:unordered_map"]]

module HR
{
    struct Employee
    {
       long   number;
       string firstName;
       string lastName;
    }
    dictionary<long, Employee> EmployeeMap;

    interface Office
    {
        ["cpp:type:std::unordered_map<Ice::Long, Employee>"] EmployeeMap
getAllEmployees();
    }
}
```

With this definition, `getAllEmployees` returns an `unordered_map`, while other unqualified parameters of type `EmployeeMap` would use the default mapping (to a `std::map`).

cpp:type with custom C++ types

If your C++ type does not look like a standard C++ container, you need to tell Ice how to marshal and unmarshal this type by providing your own `StreamHelper` specialization for this type.

For example, you can map a Slice `sequence<byte>` to a Google Protocol Buffer C++ class, `tutorial:Person`, with the following metadata directive:

Slice

```
module Demo
{
   ["cpp:type:tutorial::Person"] sequence<byte> Person;
}
```

Since `tutorial::Person` does not look like a `vector<Ice::Byte>` or a `list<Ice::Byte>`, you need a `StreamHelper` specialization for `tutorial::Person`.

The simplest is to create a `StreamHelper` specialization that handles only this specific class:

C++

```
namespace Ice
{
    template<>
    struct StreamHelper<tutorial::Person, StreamHelperCategoryUnknown>
    {
        template<class S> static inline void
        write(S* stream, const tutorial::Person& v)
        {
            // ... marshal v into a sequence of bytes...
        }

        template<class S> static inline void
        read(S* stream, tutorial::Person& v)
        {
            //... unmarshal bytes from stream into v...
        }
    };
}
```

You should also provide the corresponding `StreamTraits` specialization:

C++

```
namespace Ice
{
    template<>
    struct StreamableTraits<tutorial::Person>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryUnknown;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
}
```

This `StreamTraits` specialization is actually optional for `tutorial::Person`, and more generally for any mapping of `sequence<byte>`, since these are the values provided by the base `StreamableTraits` template.

Finally, remember to insert the header for these `StreamHelper` and `StreamTraits` specializations with a `cpp:include` global metadata directive:

Slice

```
[["cpp:include:Person.pb.h"]]
[["cpp:include:PersonStreaming.h"]]
module Demo {
["cpp:type:tutorial::Person"] sequence<byte> Person;
```

Now, if your application maps several Slice `sequence<byte>` to different Google Protocol Buffers, you could provide a `StreamHelper` specialization for each of these Google Protocol Buffer classes, but it would be more judicious to provide a single partial specialization capable of marshaling and unmarshaling any Google Protocol Buffer C++ class as a Slice `sequence<byte>` in a stream:

C++

```
namespace Ice
{
    // A new helper category for all Google Protocol Buffers
    const StreamHelperCategory StreamHelperCategoryProtobuf = 100;

    // All classes derived from ::google::protobuf::MessageLite will use
this StreamableTraits
    template<typename T>
    struct StreamableTraits<T, typename std::enable_if<std::is_base_of<
::google::protobuf::MessageLite, T>::value >::type>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryProtobuf;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };
    // T can be any Google Protocol Buffer C++ class
    template<typename T>
    struct StreamHelper<T, StreamHelperCategoryProtobuf>
    {
        template<class S> static inline void
        write(S* stream, const T& v)
        {
            std::vector<Byte> data(v.ByteSize());
            // ... marshal v into a sequence of bytes...
            stream->write(&data[0], &data[0] + data.size());
        }

        template<class S> static inline void
        read(S* stream, T& v)
        {
            std::pair<const Byte*, const Byte*> data;
            stream->read(data);
            //... unmarshal data into v...
        }
    };
}
```

If you use any of these `sequence<byte>` mapped to Google Protocol Buffers for optional data members or optional parameters, you also need to tell Ice which optional format to use:

C++

```
namespace Ice
{
    // Optional format for Slice sequence<byte> mapped to Google
Protocol Buffer
    // The template parameters correspond to the data members of the
corresponding StreamTraits specialization.
    template<>
    struct GetOptionalFormat<StreamHelperCategoryProtobuf, 1, false>
    {
        static const OptionalFormat value = OptionalFormatVSize;
    };
}
```

The `OptionalFormat` provided by `GetOptionalFormat` for `StreamHelperCategoryUnknown` and its default `StreamableTraits` is `OptionalFormatVSize`, like in the example above. This way, if your application needs a single `sequence<byte>` mapped to a custom C++ type, you can provide just a `StreamHelper` specialization for this type and rely on the default `StreamTraits` and `GetOptionalFormat` templates.

cpp:type:string and cpp:type:wstring

The metadata directives `cpp:type:string` and `cpp:type:wstring` are used map Slice `strings` to `std::string` or `std::wstring`, as described in Alternative String Mapping for C++. The Slice to C++ compiler recognize `string` and `wstring` as special tokens and does not treat them like C++ types with these names.

For example, you can map a `sequence<string>` to a `std::vector<std::wstring>` with either of the following directives:

Slice

```
// Special cpp:type:wstring metadata directive: it maps the sequence to
a std::vector<std::wstring>, not to a wstring!
["cpp:type:wstring"] sequence<string>;
```

or

Slice

```
// Maps the sequence to a std::vector<std::wstring> using a regular
cpp:type metadata directive
["cpp:type:std::vector<std::wstring>"] sequence<string>;
```

Avoiding copies with cpp:view-type

The main drawback of using standard C++ library classes to represent strings, sequences and dictionaries (as Ice does by default) is copying. For example, if we transfer an array of characters with Ice:

Slice

```
void sendChars(string s);
```

with the default mapping, our C++ code would look like:

C++

```
// client side
const char* cstring = ... null terminated array of chars;
proxy->sendChars(string(cstring));

// server side
void
sendChars(const string& s, ...)
{
   // use s;
}
```

Each invocation triggers a number of copies:

- (client) the creation of the string on the client side makes a copy of the array of characters
- (client) `sendChars` copies the characters of the string into the client-side marshaling buffer
- (server) the unmarshaling code creates a string from the bytes in the server-side marshaling buffer

If instead of `std::string`, Ice could use a string-type with view semantics–its instances point to memory owned by some other objects–we could avoid these copies. This is exactly what the `cpp:view-type` metadata directive allows us to do: select a custom mapping for Slice `strings`, `sequences` and `dictionaries`.

For example, we can map our Slice string parameter in the example above to a `std::string_view`, that has such view semantics:

Slice

```
void sendChars(["cpp:view-type:std::string_view"] string s);
```

Our C++ code pretty much the same, but we avoid several copies:

C++

```
// client side
const char* cstring = ... null terminated array of chars;
proxy->sendChars(string_view(cstring)); // string_view points to the
characters cstring, without copy

// server side
void
sendChars(const string_view& s, ...) // string_view points to bytes in
the server-side unmarshaling buffer
{
   // use s;
}
```

With this metadata directive, the only remaining copy during each invocation is:

- (client) `sendChars` copies the characters into the client-side marshaling buffer

This `cpp:view-type` metadata is very much like the `cpp:type` metadata directive described earlier, and just like for `cpp:type`, the Slice to C++ compiler uses the string provided after the `cpp:view-type:` prefix literally in the generated code. The compiler does not (and cannot) check that this string represents a valid C++ type, or a C++ type with view semantics.

There are however two significant differences between `cpp:view-type` and `cpp:type:`

- `cpp:view-type` can be applied only to operation parameters, while `cpp:type` can be applied to the definition of sequence and dictionary types, to operation parameters and to data members
- `cpp:view-type` changes the mapping of a parameter to the specified C++ type only when it is safe to use a view object (an object that does not own memory), while `cpp:type` changes the mapping all the time.

For example, if instead of sending an array of characters from a client to a server, we return an array of character from the server to the client, without using AMI or AMD (the default):

Slice

```
string getChars();
```

With the default mapping, we get a `std::string` back, and each invocation makes a few copies:

C++

```
// client-side
string s = prx->getChars();

// server side
string
getChars(...)
{
   const char* cstring = ... null terminated array of chars;
   return cstring; // the conversion to string makes a copy
}
```

Now, if we map the returned string to a `string_view` with the `cpp:type` metadata directive:

Slice

```
["cpp:type:std::string_view"] string getChars(); // don't do this
```

we avoid some copies but our `string_view` now points to deallocated memory and our program will crash!

C++

```
// client-side
string_view s = prx->getChars(); // string_view points to the
client-side marshaling buffer, which is deallocated as soon as
getChars() returns

// server side
string_view
getChars(...)
{
   const char* cstring = ... null terminated array of chars;
   return cstring; // string_view points to (or may point to)
stack-allocated memory, reclaimed when getChars() returns
}
```

> Never use the `cpp:type` metadata directive with a view type (a type that does not manage its memory); you should only use `cpp:view-type` with view types.

With `cpp:view-type`, the compiler changes the mapping only when it's safe to use a view-type, namely for:

- Input parameters, on the client-side and on the server-side
- Out and return parameters provided by the Ice run-time to AMI type-safe callbacks and AMI lambdas
- Out and return parameters provided to AMD callbacks

The `cpp:array` and `cpp:range` metadata directives for sequences follow the same rules.

With our `getChars` example:

Slice

```
["cpp:view-type:std::string_view"] string getChars();
```

it is not safe the change the client-side mapping or the server-side mapping (unless we use AMD), so the returned C++ type remains a `std::string`.

Like with the `cpp:type` metadata directive, if the C++ type specified with `cpp:view-type` is a standard library type (such as `std::list`) or looks like one, Ice will automatically marshal and unmarshal it for you. You just need to include the corresponding header with the `cpp:include` global metadata directive:

Slice

```
[["cpp:include:MyContainer.h"]]
module Sample
{
    ...
}
```

For other C++ types, you need to tell Ice how to marshal and unmarshal these types, as described above in cpp:type with custom C++ types.

In particular, Ice does not know how to marshal and unmarshal the `string_view` type. If you want to map some string parameters to `string_view`, you need to provide a `StreamHelper` for `string_view`, such as:

C++

```
namespace Ice
{
    template<>
    struct StreamableTraits<std::string_view>
    {
        static const StreamHelperCategory helper =
StreamHelperCategoryBuiltin;
        static const int minWireSize = 1;
        static const bool fixedLength = false;
    };

    template<>
    struct StreamHelper<std::string_view, StreamHelperCategoryBuiltin>
    {
        // This implementation does not perform string conversions
        template<class S> static inline void
        write(S* stream, const std::string_view& v)
        {
            stream->write(v.data(), v.size(), false);
        }
        template<class S> static inline void
        read(S* stream, std::string_view& v)
        {
            const char* vdata = 0;
            size_t vsize = 0;
            stream->read(vdata, vsize);
            v = std::string_view(vdata, vsize);
        }
    };
}
```

This `StreamHelper` specialization will take care of plain `string` parameters, and also sequence or dictionary parameters that contain `strings` mapped to `string_view` (when it's safe to do so), such as:

Slice

```
["amd", "cpp:view-type:std::vector<std::string_view>"] StringSeq
 echoStringSeq(["cpp:view-type:std::vector<std::string_view>"] StringSeq
seq);
```

The Ice/throughput demo illustrates the use of `cpp:view-type` with sequences of strings.

Using both cpp:type and cpp:view-type

If you specify both `cpp:type` and `cpp:view-type` for the same parameter, `cpp:view-type` applies to mapped parameters safe for view types, and `cpp:type` applies to all other mapped parameters.

With the following somewhat contrived example:

Slice

```
["amd", "cpp:view-type:std::vector<std::string_view>",
"cpp:type:std::list<std::wstring>"] StringSeq getStringSeq();
```

calling `getStringSeq()` on a proxy returns a `std::list<std::wstring>`, while the `StringSeq` is passed to the AMD callback as a `std::vector<string_view>`.

See Also

- Data Encoding
- Dynamic Ice

Version Information in C++98

Ice header files include the definitions of three macros that expand to the version of the Ice run time:

C++

```
#define ICE_STRING_VERSION "3.7.2" // "<major>.<minor>.<patch>"
#define ICE_INT_VERSION 30702      // AABBCC, with AA=major,
                                   // BB=minor, CC=patch

#define ICE_SO_VERSION "37"
```

`ICE_STRING_VERSION` is a string literal in the form *<major>.<minor>.<patch>*, for example, `3.7.2`. For alpha and beta releases, this string version is *<major>.<minor>[a|b]<number>*, for example, `3.7a3`.

`ICE_INT_VERSION` is an integer literal in the form *AABBCC*, where *AA* is the major version number, *BB* is the minor version number, and *CC* is the patch level, for example, `30602` for version 3.6.2. For alpha and beta releases, the patch level is set to 5x (alphas) or 6x (betas) so, for example, for version 3.7a3, the value is `30753`.

`ICE_SO_VERSION` is a string that contains the version in the soname for the Ice library. For releases, it's in the form "36" or "37", without the patch version number. For alpha and beta releases, it's identical to `ICE_STRING_VERSION` but without a dot.

slice2cpp Command-Line Options (C++98)

`slice2cpp` Command-Line Options

The Slice-to-C++ compiler, `slice2cpp`, generates C++ code for both the C++98 and the C++11 mapping, and is described on this page.

See Also

- Using the Slice Compilers

C++98 Strings and Character Encoding

On the wire, Ice transmits all strings as Unicode strings in UTF-8 encoding. For languages other than C++, Ice uses strings in their language-native Unicode representation and converts automatically to and from UTF-8 for transmission, so applications can transparently use characters from non-English alphabets.

However, for C++, how strings are represented inside a process depends on the platform as well as the mapping that is chosen for a particular string: the default mapping to `std::string`, or the alternative mapping to `std::wstring`.

> This discussion is only relevant for C++. For scripting language mappings based on Ice for C++, it is possible to use Ice's default string converter plug-in and to install your own string converter plug-in.

We will explore how strings are encoded by the Ice for C++ run time, and how you can achieve automatic conversion of strings in their native representation to and from UTF-8. For an example of using string converters in C++, refer to the sample program provided in the `demo/Ice/converter` subdirectory of your Ice distribution.

By default, the Ice run time encodes strings as follows:

- Narrow strings (that is, strings mapped to `std::string`) are presented to the application in UTF-8 encoding and, similarly, the application is expected to provide narrow strings in UTF-8 encoding to the Ice run time for transmission.

  With this default behavior, the application code is responsible for converting between the native codeset for 8-bit characters and UTF-8. For example, if the native codeset is ISO Latin-1, the application is responsible for converting between UTF-8 and narrow (8-bit) characters in ISO Latin-1 encoding.

  Also note that the default behavior does not require the application to do anything if it only uses characters in the ASCII range. (This is because a string containing only characters in the (7-bit) ASCII range is also a valid UTF-8 string.)

- Wide strings (that is, strings mapped to `std::wstring`) are automatically encoded as Unicode by the Ice run time as appropriate for the platform. For example, for Windows, the Ice run time converts between UTF-8 and UTF-16 in little-endian representation whereas, for Linux, the Ice run time converts between UTF-8 and UTF-32 in the endian-ness appropriate for the host CPU.

  With this default behavior, wide strings are transparently converted between their on-the-wire representation and their native C++ representation as appropriate, so application code need not do anything special. (The exception is if an application uses a non-Unicode encoding, such as Shift-JIS, as its native `wstring` codeset.)

Topics

See Also

Installing String Converters with C++98

The default behavior of the run time can be changed by providing application-specific string converters. If you install such converters, all Slice strings will be passed to the appropriate converter when they are marshaled and unmarshaled. Therefore, the string converters allow you to convert all strings transparently into their native representation without having to insert explicit conversion calls whenever a string crosses a Slice interface boundary.

You can install string converters by calling `Ice::setProcessStringConverter` for the narrow string converter, and `Ice::setProcessWstringConverter` for the wide string converter. Any strings that use the default (`std::string`) mapping are passed through the specified narrow string converter and any strings that use the wide (`std::wstring`) mapping are passed through the specified wide string converter.

You can also retrieve the previously installed string converters (or default string converters) with `Ice::getProcessStringConverter` and `Ice::getProcessWstringConverter`. The default narrow string converter is null, meaning all `std::string`s use the UTF-8 encoding.

The string converters are defined as follows:

C++

```
namespace Ice
{
    class UTF8Buffer
    {
    public:
        virtual Byte* getMoreBytes(size_t howMany,
Byte* firstUnused) = 0;
        virtual ~UTF8Buffer() {}
    };

    template<typename charT>
    class BasicStringConverter : public IceUtil::Shared
    {
    public:
        virtual Byte*
toUTF8(const charT* sourceStart, const charT* sourceEnd,
                             UTF8Buffer&) const = 0;

        virtual void fromUTF8(const Byte* sourceStart,
                              const Byte* sourceEnd,
                              std::basic_string<charT>& target) const;
    };

    typedef BasicStringConverter<char> StringConverter;
    typedef IceUtil::Handle<StringConverter> StringConverterPtr;

    typedef BasicStringConverter<wchar_t> WstringConverter;
    typedef IceUtil::Handle<WstringConverter> WstringConverterPtr;

    StringConverterPtr getProcessStringConverter();
    void setProcessStringConverter(const StringConverterPtr&);

    WstringConverterPtr getProcessWstringConverter();
    void setProcessWstringConverter(const WstringConverterPtr&);
}
```

As you can see, both narrow and wide string converters are simply templates with either a narrow or a wide character (`char` or `wchar_t`) as the template parameter.

Each communicator caches the narrow string converter and wide string converter installed when this communicator is initialized.

You should always install your string converters before creating your communicator(s). When using a plugin to set your string converters, you need to set the string converters in the constructor of your plugin class (which is executed when the plugin is loaded) and not in the initialization function of the plugin class (which is executed after the communicator has read and cached the process-wide string converters).

See Also

- Communicator Initialization

- C++98 Mapping for Built-In Types

UTF-8 Conversion with C++98

On this page:

Converting to UTF-8

If you have installed a string converter, the Ice run time calls the converter's `toUTF8` function whenever it needs to convert a native string into UTF-8 representation for transmission. The `sourceStart` and `sourceEnd` pointers point at the first byte and one-beyond-the-last byte of the source string, respectively. The implementation of `toUTF8` must return a pointer to the first unused byte following the converted string.

Your implementation of `toUTF8` must allocate the returned string by calling the `getMoreBytes` member function of the `UTF8Buffer` class that is passed as the third argument. (`getMoreBytes` throws a `std::bad_alloc` if it cannot allocate enough memory.) The `firstUnused` parameter must point at the first unused byte of the allocated memory region. You can make several calls to `getMoreBytes` to incrementally allocate memory for the converted string. If you do, `getMoreBytes` may relocate the buffer in memory. (If it does, it copies the part of the string that was converted so far into the new memory region.) The function returns a pointer to the first unused byte of the (possibly relocated) memory.

Conversion can also fail because the encoding of the source string is internally incorrect. In that case, you should throw a `Ice::IllegalConversionException` exception from `toUTF8`, for example:

C++

```
throw Ice::IllegalConversionException(__FILE__, __LINE__, "bad encoding
because ...");
```

After it has marshaled the returned string into an internal marshaling buffer, the Ice run time deallocates the string.

Converting from UTF-8

During unmarshaling, the Ice run time calls the `fromUTF8` member function on the corresponding string converter. The function converts a UTF-8 byte sequence into its native form as a `std::string` or `std::wstring`. The string into which the function must place the converted characters is passed to `fromUTF8` as the `target` parameter.

See Also

- Installing String Converters with C++98

String Parameters in Local C++98 Calls

In C++, and indirectly in Python, Ruby, and PHP, all Ice local APIs are narrow-string based, meaning you could not for example recompile `Properties.ice` to get property names and values as wide strings.

Installing a narrow-string converter could cause trouble for these local calls if UTF-8 conversion occurs in the underlying implementation. For example, the `stringToIdentity` operation creates an intermediary UTF-8 string. If this string contains characters that are not in your native codeset (as determined by the narrow-string converter), the `stringToIdentity` call will fail.

Likewise, when Ice reads properties from a configuration file, it converts the input (UTF-8 characters) into native strings. This conversion can also fail if the native encoding cannot convert some characters.

Most strings in local calls are never problematic because Ice does not perform any conversion, for example:

- adapter names in `createObjectAdapter`
- property names and values in `Properties`
- `ObjectAdapter::createProxy`, where the identity conversion occurs only when the proxy is marshaled

Finally, consider the Slice type `Ice::Context`, which is mapped in C++ as a `map<string, string>`. The mapping for `Context` cannot be changed to `map<wstring, wstring>`, therefore you cannot send or receive any context entry that is not in your narrow-string native encoding when a narrow-string converter is installed.

See Also

- Object Identity
- Properties and Configuration
- Request Contexts

Built-in String Converters in C++98

Ice provides three string converters to cover common conversion requirements:

- The Unicode wstring converter
  This is a string converter that converts between Unicode wide strings and UTF-8 strings. Unless you install a different string converter, this is the default converter that is used for wide strings.
- The iconv string converter (Linux, macOS, Unix)
  The `iconv` string converter converts strings using the `iconv` conversion facility. It can be used to convert either wide or narrow strings.
- The Windows string converter (Windows only)
  This string converter converts between multi-byte and UTF-8 strings and uses `MultiByteToWideChar` and `WideCharToMultiByte` for its implementation.

These string converters can be created through factory functions in the `Ice` namespace:

C++

```
namespace Ice
{
    WstringConverterPtr createUnicodeWstringConverter();

    template<typename charT>
    IceUtil::Handle<BasicStringConverter<charT>>
createIconvStringConverter(const std::string& internalCode = "");

    StringConverterPtr createWindowsStringConverter(unsigned int
codepage);
}
```

See Also

- The C++98 iconv String Converter

C++98 String Conversion Convenience Functions

Ice provides two helper functions that allow you to convert between narrow strings and UTF-8 encoded strings using a string converter:

C++

```
namespace Ice
{
    std::string nativeToUTF8(const std::string&, const
StringConverterPtr&);
    std::string UTF8ToNative(const std::string&, const
StringConverterPtr&);
}
```

No conversion is performed when the provided string converter is null.

Ice provides two additional helper functions that allow you to convert between wide strings and narrow strings using the provided string converters:

C++

```
namespace Ice
{
    std::string wstringToString(const std::wstring&, const
StringConverterPtr& = 0, const WstringConverterPtr& = 0);
    std::wstring stringToWstring(const std::string&, const
StringConverterPtr& = 0, const WstringConverterPtr& = 0);
}
```

When the narrow string converter given to `wstringToString` is null, the encoding of the returned narrow string is UTF-8. When the wide string converter given to `wstringToString` is null, the encoding of wide string parameter is UTF-16 or UTF-32, depending on the size of `wchar_t`.

Likewise for `stringToWstring`, when the wide string converter is null, the encoding of the returned wide string is UTF-16 or UTF-32 depending on the size of wchar_t. When the narrow string converter given to `stringToWstring` is null, the encoding of narrow string parameter is UTF-8.

See Also

- UTF-8 Conversion with C++98

The C++98 iconv String Converter

For Linux, macOS and Unix platforms, `Ice` provides a string converter implementation that uses the `iconv` conversion facility to convert between the native encoding and UTF-8.

To use this string converter, you specify whether the conversion you want is for narrow or wide characters via the template argument, and you specify the corresponding native encoding with the constructor argument. For example, to create a converter that converts between ISO Latin-1 and UTF-8, you can instantiate the converter as follows:

C++

```
StringConverterPtr
stringConverter = Ice::createIconvStringConverter<char>("ISO-8859-1");
```

Similarly, to convert between the internal wide character encoding and UTF-8, you can instantiate a converter as follows:

C++

```
WstringConverterPtr
wstringConverter = Ice::createIconvStringConverter<wchar_t>("WCHAR_T");
```

The string you pass to the factory function must be one of the values returned by `iconv -l`, which lists all the available character encodings for your machine. Passing no parameter or an empty string is equivalent to passing nl_langinfo(CODESET).

See Also

- Installing String Converters with C++98

The C++98 Ice String Converter Plug-in

The Ice run time includes a plug-in that supports conversion between UTF-8 and native encodings on Unix and Windows platforms. You can use this plug-in to install converters for narrow and wide strings into the communicator of an existing program. This feature is primarily intended for use in scripting language extensions such as Ice for Python[1]; if you need to use string converters in your C++ application, we recommend using the technique described in Installing String Converters with C++98 instead.

Note that an application must be designed to operate correctly in the presence of a string converter. A string converter assumes that it converts strings in the native encoding into the UTF-8 encoding, and vice versa. An application that performs its own conversions on strings that cross a Slice interface boundary can cause encoding errors when those strings are processed by a converter.

Configuring the Ice String Converter Plug-in

You can install the plug-in using a configuration property like the one shown below:

```
Ice.Plugin.IceStringConverter=Ice:createStringConverter
iconv=encoding[,encoding] windows=code-page
```

The first component of the property value represents the plug-in's entry point, which includes the abbreviated name of the shared library or DLL (`Ice`) and the name of a factory function (`createStringConverter`).

The plug-in accepts the following arguments:

- `iconv=encoding[,encoding]`
  This argument is optional on Unix platforms and ignored on Windows platforms. If specified, it defines the `iconv` names of the narrow string encoding and the optional wide-string encoding. If this argument is not specified, the plug-in installs a narrow string converter that uses the default locale-dependent encoding.
- `windows=code-page`
  This argument is required on Windows platforms and ignored on Unix platforms. The *`code-page`* value represents a code page number, such as `1252`.

The plug-in's argument semantics are designed so that the same configuration property can be used on both Windows and Unix platforms, as shown in the following example:

```
Ice.Plugin.IceStringConverter=Ice:createStringConverter
iconv=ISO8859-1 windows=1252
```

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.IceStringConverter.cpp=Ice:createStringConverter
iconv=ISO8859-1 windows=1252
```

If using static libraries, you must also call the `Ice::registerIceStringConverter` function to ensure the plug-in is linked with your application.

[1] Ice for Python only supports string converters when using Python 2.x.

See Also

- UTF-8 Conversion with C++98
- Installing String Converters with C++98
- Plug-in Configuration
- Ice.InitPlugins
- Ice.Plugin.*
- Ice.PluginLoadOrder

Custom String Converter Plug-ins with C++98

If the default string converter plug-in does not satisfy your requirements, you can install your own string converters with a plug-in, for example:

C++

```
class MyStringConverterPlugin : public Ice::Plugin
{
public:

    MyStringConverterPlugin(const Ice::StringConverterPtr&
stringConverter,

const Ice::WstringConverterPtr& wstringConverter = 0)
    {
        setProcessStringConverter(stringConverter);
        setProcessWstringConverter(wstringConverter);
    }

    virtual void initialize() {}
    virtual void destroy() {}
};
```

Like in this example, you should install the string converters in your plug-in's constructor. Do not install the string converters in `initialize`.

In order to create such a plug-in, you must do the following:

- Define and export a factory function that returns an instance of your plug-in class.
- Implement the string converter(s) that you will pass to your plug-in's constructor, or use the ones included with Ice.
- Package your code into a shared library or DLL.

To install your plug-in, use a configuration property like the one shown below:

```
Ice.Plugin.MyConverterPlugin=myconverter:createConverter ...
```

The first component of the property value represents the plug-in's entry point, which includes the abbreviated name of the shared library or DLL (`myconverter`) and the name of a factory function (`createConverter`).

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.MyConverterPlugin.cpp=myconverter:createConverter ...
```

See Also

- The C++98 Ice String Converter Plug-in
- Installing String Converters with C++98
- Plug-in API
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

The C++98 Utility Library

Before C++11, the standard C++ library lacked support for smart pointers with shared semantics, and did not provide a standard API to create threads, mutexes and other synchronization primitives.

Ice provides its own implementation of such utility-classes for the Ice C++98 mapping in the `IceUtil` namespace. These classes are visible only when using the C++98 mapping.

Topics

Threads and Concurrency with C++

Threading and concurrency control vary widely with different operating systems. To make threads programming easier and portable, Ice provides a simple thread abstraction layer that allows you to write portable source code regardless of the underlying platform.

This section looks at the threading and concurrency control mechanisms in Ice for C++. It explains the threading abstractions provided by Ice: mutexes, monitors, and threads. Using these APIs allows you to make your code thread safe and to create threads of your own without having to use non-portable APIs that differ in syntax or semantics across different platforms: Ice not only provides a portable API but also guarantees that the semantics of the various functions are the same across different platforms. This makes it easier to create thread-safe applications and lets you move your code between platforms with simple recompilation.

> This section assumes you are familiar with light-weight threads and concurrency control. Also see The Ice Threading Model, which provides a language-neutral introduction to the Ice threading model.

Library Overview

The Ice threading library provides the following thread-related abstractions:

- mutexes
- recursive mutexes
- monitors
- a thread abstraction that allows you to create, control, and destroy threads

The synchronization primitives permit you to implement concurrency control at different levels of granularity. In addition, the thread abstraction allows you to, for example, create a separate thread that can respond to GUI or other asynchronous events. All of the threading APIs are part of the `IceUtil` namespace.

Topics

The C++ Mutex Class

This page describes how to use mutexes — one of the available synchronization primitives.

On this page:

Mutex Member Functions

The class `IceUtil::Mutex` (defined in `IceUtil/Mutex.h`) provides a simple non-recursive mutual exclusion mechanism:

C++

```
namespace IceUtil
{
    enum MutexProtocol { PrioInherit, PrioNone };

    class Mutex
    {
    public:
        Mutex();
        Mutex(MutexProtocol p);
        ~Mutex();

        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };
}
```

The member functions of this class work as follows:

- `Mutex()`
  `Mutex(MutexProtocol p)`
  You can optionally specify a mutex protocol when you construct a mutex. The mutex protocol controls how the mutex behaves with respect to thread priorities. Default-constructed mutexes use a system-wide default.
- `lock`
  The `lock` function attempts to acquire the mutex. If the mutex is already locked, it suspends the calling thread until the mutex becomes available. The call returns once the calling thread has acquired the mutex.
- `tryLock`
  The `tryLock` function attempts to acquire the mutex. If the mutex is available, the call returns true with the mutex locked. Otherwise, if the mutex is locked by another thread, the call returns false.
- `unlock`
  The `unlock` function unlocks the mutex.

Note that `IceUtil::Mutex` is a non-recursive mutex implementation. This means that you must adhere to the following rules:

- Do not call `lock` on the same mutex more than once from a thread. The mutex is not recursive so, if the owner of a mutex attempts to lock it a second time, the behavior is undefined.

- Do not call `unlock` on a mutex unless the calling thread holds the lock. Calling `unlock` on a mutex that is not currently held by any thread, or calling `unlock` on a mutex that is held by a different thread, results in undefined behavior.

Use the `IceUtil::RecMutex` class if you need recursive semantics.

Adding Thread Safety to the File System Application in C++

Recall that the implementation of the `read` and `write` operations for our file system server is not thread safe:

C++

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    return _lines;      // Not thread safe!
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
    _lines = text;      // Not thread safe!
}
```

The problem here is that, if we receive concurrent invocations of `read` and `write`, one thread will be assigning to the `_lines` vector while another thread is reading that same vector. The outcome of such concurrent data access is undefined; to avoid the problem, we need to serialize access to the `_lines` member with a mutex. We can make the mutex a data member of the `FileI` class and lock and unlock it in the `read` and `write` operations:

C++

```
#include <IceUtil/Mutex.h>
// ...

namespace Filesystem
{
    // ...

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI
    {
    public:
        // As before...
    private:
        Lines _lines;
        IceUtil::Mutex _fileMutex;
    };
    // ...
}

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    _fileMutex.lock();
    Lines l = _lines;
    _fileMutex.unlock();
    return l;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
const Ice::Current&)
{
    _fileMutex.lock();
    _lines = text;
    _fileMutex.unlock();
}
```

The `FileI` class here is identical to the original implementation, except that we have added the `_fileMutex` data member. The `read` and `write` operations lock and unlock the mutex to ensure that only one thread can read or write the file at a time. Note that, by using a separate mutex for each `FileI` instance, it is still possible for multiple threads to concurrently read or write files, as long as they each access a *different* file. Only concurrent accesses to the *same* file are serialized.

The implementation of `read` is somewhat awkward here: we must make a local copy of the file contents while we are holding the lock and return that copy. Doing so is necessary because we must unlock the mutex before we can return from the function. However, as we will see in the next section, the copy can be avoided by using a helper class that unlocks the mutex automatically when the function returns.

Guaranteed Unlocking of Mutexes in C++

Using the raw `lock` and `unlock` operations on mutexes has an inherent problem: if you forget to unlock a mutex, your program will

deadlock. Forgetting to unlock a mutex is easier than you might suspect, for example:

C++

```
Filesystem::Lines
Filesystem::File::read(const Ice::Current&) const
{
    _fileMutex.lock();                  // Lock the mutex
    Lines l = readFileContents();       // Read from database
    _fileMutex.unlock();                // Unlock the mutex
    return l;
}
```

Assume that we are keeping the contents of the file on secondary storage, such as a database, and that the `readFileContents` function accesses the file. The code is almost identical to the previous example but now contains a latent bug: if `readFileContents` throws an exception, the `read` function terminates without ever unlocking the mutex. In other words, this implementation of `read` is not exception-safe.

The same problem can easily arise if you have a larger function with multiple return paths. For example:

C++

```
void
SomeClass::someFunction(/* params here... */)
{
    _mutex.lock();                      // Lock a mutex

    // Lots of complex code here...

    if(someCondition)
    {
        // More complex code here...
        return;                         // Oops!!!
    }

    // More code here...

    _mutex.unlock();                    // Unlock the mutex
}
```

In this example, the early return from the middle of the function leaves the mutex locked. Even though this example makes the problem quite obvious, in large and complex pieces of code, both exceptions and early returns can cause hard-to-track deadlock problems. To avoid this, the `Mutex` class contains two type definitions for helper classes, called `Lock` and `TryLock`:

C++

```
namespace IceUtil
{
    class Mutex
    {
        // ...

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };
}
```

`LockT` and `TryLockT` are simple templates that primarily consist of a constructor and a destructor; the `LockT` constructor calls `lock` on its argument, and the `TryLockT` constructor calls `tryLock` on its argument. The destructors call `unlock` if the mutex is locked when the template goes out of scope. By instantiating a local variable of type `Lock` or `TryLock`, we can avoid the deadlock problem entirely:

C++

```
void
SomeClass::someFunction(/* params here... */)
{
    IceUtil::Mutex::Lock lock(_mutex);  // Lock a mutex

    // Lots of complex code here...

    if(someCondition)
    {
        // More complex code here...
        return;                         // No problem
    }

    // More code here...

}   // Destructor of lock unlocks the mutex
```

This is an example of the *RAII* (*Resource Acquisition Is Initialization*) idiom [1].

On entry to `someFunction`, we instantiate a local variable `lock`, of type `IceUtil::Mutex::Lock`. The constructor of `lock` calls `lock` on the mutex so the remainder of the function is inside a critical region. Eventually, `someFunction` returns, either via an ordinary return (in the middle of the function or at the end) or because an exception was thrown somewhere in the function body. Regardless of how the function terminates, the C++ run time unwinds the stack and calls the destructor of `lock`, which unlocks the mutex, so we cannot get trapped by the deadlock problem we had previously.

Both the `Lock` and `TryLock` templates have a few member functions:

- `void acquire() const`
  This function attempts to acquire the lock and blocks the calling thread until the lock becomes available. If the caller calls `acquire` on a mutex it has locked previously, the function throws `ThreadLockedException`.
- `bool tryAcquire() const`

This function attempts to acquire the mutex. If the mutex can be acquired, it returns true with the mutex locked; if the mutex cannot be acquired, it returns false. If the caller calls `tryAcquire` on a mutex it has locked previously, the function throws `ThreadLockedException`.

- `void release() const`
  This function releases a previously locked mutex. If the caller calls release on a mutex it has unlocked previously, the function throws `ThreadLockedException`.
- `bool acquired() const`
  This function returns true if the caller has locked the mutex previously, otherwise it returns false. If you use the `TryLock` template, you must call `acquired` after instantiating the template to test whether the lock actually was acquired.

These functions are useful if you want to use the `Lock` and `TryLock` templates for guaranteed unlocking, but need to temporarily release the lock:

C++

```
{
    IceUtil::Mutex::TryLock m(someMutex);

    if(m.acquired())
    {

        // Got the lock, do processing here...

        if(release_condition)
        {
            m.release();
        }

        // Mutex is now unlocked, someone else can lock it.
        // ...

        m.acquire(); // Block until mutex becomes available.

        // ...

        if(release_condition)
        {
            m.release();
        }

        // Mutex is now unlocked, someone else can lock it.

        // ...

        // Spin on the mutex until it becomes available.
        while(!m.tryLock())
        {
            // Do some other processing here...
        }

        // Mutex locked again at this point.

        // ...
    }

} // Close scope, m is unlocked by its destructor.
```

Tip
You should make it a habit to always use the `Lock` and `TryLock` helpers instead of calling `lock` and `unlock` directly. Doing so results in code that is easier to understand and maintain.

Using the `Lock` helper, we can rewrite the implementation of our `read` and `write` operations as follows:

C++

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
const Ice::Current&)
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    _lines = text;
}
```

Note that this also eliminates the need to make a copy of the `_lines` data member: the return value is initialized under protection of the mutex and cannot be modified by another thread once the destructor of `lock` unlocks the mutex.

See Also

- Example of a File System Server in C++98
- Priority Inversion in C++
- The C++ RecMutex Class

References

1. Stroustrup, B. 1997. *The C++ Programming Language*. Reading, MA: Addison-Wesley.

The C++ RecMutex Class

A non-recursive mutex cannot be locked more than once, even by the thread that holds the lock. This frequently becomes a problem if a program contains a number of functions, each of which must acquire a mutex, and you want to call one function as part of the implementation of another function:

C++

```
IceUtil::Mutex _mutex;

void
f1()
{
    IceUtil::Mutex::Lock lock(_mutex);
    // ...
}

void
f2()
{
    IceUtil::Mutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                                // Deadlock!

    // More code here...
}
```

`f1` and `f2` each correctly lock the mutex before manipulating data but, as part of its implementation, `f2` calls `f1`. At that point, the program deadlocks because `f2` already holds the lock that `f1` is trying to acquire. For this simple example, the problem is obvious. However, in complex systems with many functions that acquire and release locks, it can get very difficult to track down this kind of situation: the locking conventions are not manifest anywhere but in the source code and each caller must know which locks to acquire (or not to acquire) before calling a function. The resulting complexity can quickly get out of hand.

Ice provides a recursive mutex class `RecMutex` (defined in `IceUtil/RecMutex.h`) that avoids this problem:

C++

```
namespace IceUtil
{
    class RecMutex
    {
    public:
        RecMutex();
        RecMutex(MutexProtocol p);
        ~RecMutex();

        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<RecMutex> Lock;
        typedef TryLockT<RecMutex> TryLock;
    };
}
```

Note that the signatures of the operations are the same as for `IceUtil::Mutex`. However, `RecMutex` implements a recursive mutex:

- `RecMutex()`
  `RecMutex(MutexProtocol p)`
  You can optionally specify a mutex protocol when you construct a mutex. The mutex protocol controls how the mutex behaves with respect to thread priorities. Default-constructed mutexes use a system-wide default.
- `lock`
  The `lock` function attempts to acquire the mutex. If the mutex is already locked by another thread, it suspends the calling thread until the mutex becomes available. If the mutex is available or is already locked by the calling thread, the call returns immediately with the mutex locked.
- `tryLock`
  The `tryLock` function works like `lock`, but, instead of blocking the caller, it returns false if the mutex is locked by another thread. Otherwise, the return value is true.
- `unlock`
  The `unlock` function unlocks the mutex.

As for non-recursive mutexes, you must adhere to a few simple rules for recursive mutexes:

- Do not call `unlock` on a mutex unless the calling thread holds the lock.
- You must call `unlock` as many times as you called `lock` for the mutex to become available to another thread. (Internally, a recursive mutex is implemented with a counter that is initialized to zero. Each call to `lock` increments the counter and each call to `unlock` decrements the counter; the mutex is made available to another thread when the counter returns to zero.)

Using recursive mutexes, the code fragment shown earlier works correctly:

C++

```
#include <IceUtil/RecMutex.h>
// ...

IceUtil::RecMutex _mutex;       // Recursive mutex

void
f1()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // ...
}

void
f2()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                               // Fine

    // More code here...
}
```

Note that the type of the mutex is now `RecMutex` instead of `Mutex`, and that we are using the `Lock` type definition provided by the `RecMutex` class, not the one provided by the `Mutex` class.

See Also

- The C++ Mutex Class
- Priority Inversion in C++

The C++ Monitor Class

The recursive and non-recursive mutex classes implement a simple mutual exclusion mechanism that allows only a single thread to be active in a critical region at a time. In particular, for a thread to enter the critical region, another thread must leave it. This means that, with mutexes, it is impossible to suspend a thread inside a critical region and have that thread wake up again at a later time, for example, when a condition becomes true.

To address this problem, Ice provides a monitor. Briefly, a monitor is a synchronization mechanism that protects a critical region: as for a mutex, only one thread may be active at a time inside the critical region. However, a monitor allows you to suspend a thread inside the critical region; doing so allows another thread to enter the critical region. The second thread can either leave the monitor (thereby unlocking the monitor), or it can suspend itself inside the monitor; either way, the original thread is woken up and continues execution inside the monitor. This extends to any number of threads, so several threads can be suspended inside a monitor.

> The monitors provided by Ice have *Mesa* semantics, so called because they were first implemented by the Mesa programming language [1]. Mesa monitors are provided by a number of languages, including Java and Ada. With Mesa semantics, the signalling thread continues to run and another thread gets to run only once the signalling thread suspends itself or leaves the monitor.

Monitors provide a more flexible mutual exclusion mechanism than mutexes because they allow a thread to check a condition and, if the condition is false, put itself to sleep; the thread is woken up by some other thread that has changed the condition.

On this page:

Monitor Member Functions

Ice provides monitors with the `IceUtil::Monitor` class (defined in `IceUtil/Monitor.h`):

C++

```
namespace IceUtil
{
    template <class T>
    class Monitor
    {
    public:
        void lock() const;
        void unlock() const;
        bool tryLock() const;

        void wait() const;
        bool timedWait(const Time&) const;
        void notify();
        void notifyAll();

        typedef LockT<Monitor<T> > Lock;
        typedef TryLockT<Monitor<T> > TryLock;
    };
}
```

Note that `Monitor` is a template class that requires either `Mutex` or `RecMutex` as its template parameter. (Instantiating a `Monitor` with a `RecMutex` makes the monitor recursive.)

The member functions behave as follows:

- `lock`
  This function attempts to lock the monitor. If the monitor is currently locked by another thread, the calling thread is suspended until the monitor becomes available. The call returns with the monitor locked.
- `tryLock`
  This function attempts to lock a monitor. If the monitor is available, the call returns true with the monitor locked. If the monitor is locked by another thread, the call returns false.
- `unlock`
  This function unlocks a monitor. If other threads are waiting to enter the monitor (are blocked inside a call to `lock`), one of the threads is woken up and locks the monitor.
- `wait`
  This function suspends the calling thread and, at the same time, releases the lock on the monitor. A thread suspended inside a call to `wait` can be woken up by another thread that calls `notify` or `notifyAll`. When the call returns, the suspended thread resumes execution with the monitor locked.
- `timedWait`
  This function suspends the calling thread for up to the specified timeout. If another thread calls `notify` or `notifyAll` and wakes up the suspended thread before the timeout expires, the call returns true and the suspended thread resumes execution with the monitor locked. Otherwise, if the timeout expires, the function returns false. Wait intervals are represented by instances of the `Time` class.
- `notify`
  This function wakes up a single thread that is currently suspended in a call to `wait` or `timedWait`. If no thread is suspended in a call to `wait` or `timedWait` at the time `notify` is called, the notification is lost (that is, calls to `notify` are *not* remembered if there is no thread to be woken up). Note that notifying does not run another thread immediately. Another thread gets to run only once the notifying thread either calls `wait` or `timedWait` or unlocks the monitor (Mesa semantics).
- `notifyAll`
  This function wakes up all threads that are currently suspended in a call to `wait` or `timedWait`. As for `notify`, calls to `notifyAll` are lost if no threads are suspended at the time. Also as for `notify`, `notifyAll` causes other threads to run only once the notifying thread has either called `wait` or `timedWait` or unlocked the monitor (Mesa semantics).

You must adhere to a few rules for monitors to work correctly:

- Do not call `unlock` unless you hold the lock. If you instantiate a monitor with a recursive mutex, you get recursive semantics, that is, you must call `unlock` as many times as you have called `lock` (or `tryLock`) for the monitor to become available.
- Do not call `wait` or `timedWait` unless you hold the lock.
- Do not call `notify` or `notifyAll` unless you hold the lock.
- When returning from a `wait` call, you *must* re-test the condition before proceeding (as shown below).

Using Monitors in C++

To illustrate how to use a monitor, consider a simple unbounded queue of items. A number of producer threads add items to the queue, and a number of consumer threads remove items from the queue. If the queue becomes empty, consumers must wait until a producer puts a new item on the queue. The queue itself is a critical region, that is, we cannot allow a producer to put an item on the queue while a consumer is removing an item. Here is a very simple implementation of a such a queue:

C++

```
template<class T> class Queue
{
public:
    void put(const T& item)
    {
        _q.push_back(item);
    }

    T get()
    {
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

As you can see, producers call the `put` method to enqueue an item, and consumers call the `get` method to dequeue an item. Obviously, this implementation of the queue is not thread-safe and there is nothing to stop a consumer from attempting to dequeue an item from an empty queue.

Here is a version of the queue that uses a monitor to suspend a consumer if the queue is empty:

C++

```
#include <IceUtil/Monitor.h>

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    void put(const T& item)
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        notify();
    }

    T get()
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

Note that the `Queue` class now inherits from `IceUtil::Monitor<IceUtil::Mutex>`, that is, `Queue` *is-a* monitor.

Both the `put` and `get` methods lock the monitor when they are called. As for mutexes, instead of calling `lock` and `unlock` directly, we are using the `Lock` helper which automatically locks the monitor when it is instantiated and unlocks the monitor again when it is destroyed.

The `put` method first locks the monitor and then, now being in sole possession of the critical region, enqueues an item. Before returning (thereby unlocking the monitor), `put` calls `notify`. The call to `notify` will wake up any consumer thread that may be asleep in a `wait` call to inform the consumer that an item is available.

The `get` method also locks the monitor and then, before attempting to dequeue an item, tests whether the queue is empty. If so, the consumer calls `wait`. This suspends the consumer inside the `wait` call and unlocks the monitor, so a producer can enter the monitor to enqueue an item. Once that happens, the producer calls `notify`, which causes the consumer's `wait` call to complete, with the monitor again locked for the consumer. The consumer now dequeues an item and returns (thereby unlocking the monitor).

For this machinery to work correctly, the implementation of `get` does two things:

- `get` tests whether the queue is empty *after* acquiring the lock.
- `get` re-tests the condition in a loop around the call to `wait`; if the queue is still empty after `wait` returns, the `wait` call is re-entered.

You *must* always write your code to follow the same pattern:

- *Never* test a condition unless you hold the lock.
- *Always* re-test the condition in a loop around `wait`. If the test still shows the wrong outcome, call `wait` again.

Not adhering to these conditions will eventually result in a thread accessing shared data when it is not in its expected state, for the following reasons:

1. If you test a condition without holding the lock, there is nothing to prevent another thread from entering the monitor and changing its state before you can acquire the lock. This means that, by the time you get around to locking the monitor, the state of the monitor

may no longer be in agreement with the result of the test.
2. Some thread implementations suffer from a problem known as *spurious wake-up*: occasionally, more than one thread may wake up in response to a call to `notify`, or a thread may wake up without any call to `notify` at all. As a result, each thread that returns from a call to `wait` must re-test the condition to ensure that the monitor is in its expected state: the fact that `wait` returns does *not* indicate that the condition has changed.

Efficient Notification using Monitors in C++

The previous implementation of our thread-safe queue unconditionally notifies a waiting reader whenever a writer deposits an item into the queue. If no reader is waiting, the notification is lost and does no harm. However, unless there is only a single reader and writer, many notifications will be sent unnecessarily, causing unwanted overhead.

Here is one way to fix the problem:

C++

```
#include <IceUtil/Monitor.h>

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    void put(const T& item)
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if(_q.size() == 1)
        {
            notify();
        }
    }

    T get()
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while(_q.size() == 0)
        {
            wait();
        }
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

The only difference between this code and the implementation shown earlier is that a writer calls `notify` only if the queue length has just changed from empty to non-empty. That way, unnecessary `notify` calls are never made. However, this approach works only for a single reader thread. To see why, consider the following scenario:

1. Assume that the queue currently contains a number of items and that we have five reader threads.
2. The five reader threads continue to call `get` until the queue becomes empty and all five readers are waiting in `get`.
3. The scheduler schedules a writer thread. The writer finds the queue empty, deposits an item, and wakes up a single reader thread.
4. The awakened reader thread dequeues the single item on the queue.

5. The reader calls `get` a second time, finds the queue empty, and goes to sleep again.

The net effect of this is that there is a good chance that only one reader thread will ever be active; the other four reader threads end up being permanently asleep inside the `get` method.

One way around this problem is call `notifyAll` instead of `notify` once the queue length exceeds a certain amount, for example:

C++

```
#include <IceUtil/Monitor.h>

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    void put(const T& item)
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if(_q.size() >= _wakeupThreshold)
        {
            notifyAll();
        }
    }

    T get()
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while(_q.size() == 0)
        {
            wait();
        }
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    const int _wakeupThreshold = 100;
};
```

Here, we have added a private data member `_wakeupThreshold`; a writer wakes up *all* waiting readers once the queue length exceeds the threshold, in the expectation that all the readers will consume items more quickly than they are produced, thereby reducing the queue length below the threshold again.

This approach works, but has drawbacks as well:

- The appropriate value of `_wakeupThreshold` is difficult to determine and sensitive to things such as speed and number of processors and I/O bandwidth.
- If multiple readers are asleep, they are all made runnable by the thread scheduler once a writer calls `notifyAll`. On a multiprocessor machine, this may result in all readers running at once (one per CPU). However, as soon as the readers are made runnable, each of them attempts to reacquire the mutex that protects the monitor before returning from `wait`. Of course, only one of the readers actually succeeds and the remaining readers are suspended again, waiting for the mutex to become available. The net result is a large number of thread context switches as well as repeated and unnecessary locking of the system bus.

A better option than calling `notifyAll` is to wake up waiting readers one at a time. To do this, we keep track of the number of waiting readers and call `notify` only if a reader needs to be woken up:

C++

```
#include <IceUtil/Monitor.h>

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    Queue() : _waitingReaders(0) {}

    void put(const T& item)
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if(_waitingReaders)
        {
            notify();
        }
    }

    T get()
    {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while(_q.size() == 0)
        {
            try
            {
                ++_waitingReaders;
                wait();
                --_waitingReaders;
            }
            catch (...)
            {
                --_waitingReaders;
                throw;
            }
        }
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    short _waitingReaders;
};
```

This implementation uses a member variable `_waitingReaders` to keep track of the number of readers that are suspended. The

constructor initializes the variable to zero and the implementation of `get` increments and decrements the variable around the call to `wait`. Note that these statements are enclosed in a `try-catch` block; this ensures that the count of waiting readers remains accurate even if `wait` throws an exception. Finally, `put` calls `notify` only if there is a waiting reader.

The advantage of this implementation is that it minimizes contention on the monitor mutex: a writer wakes up only a single reader at a time, so we do not end up with multiple readers simultaneously trying to lock the mutex. Moreover, the monitor `notify` implementation signals a waiting thread only *after* it has unlocked the mutex. This means that, when a thread wakes up from its call to `wait` and tries to reacquire the mutex, the mutex is likely to be unlocked. This results in more efficient operation because acquiring an unlocked mutex is typically very efficient, whereas forcefully putting a thread to sleep on a locked mutex is expensive (because it forces a thread context switch).

See Also

- The C++ Mutex Class
- The C++ RecMutex Class
- The C++ Time Class

References

1. Mitchell, J. G., et al. 1979. *Mesa Language Manual*. CSL-793. Palo Alto, CA: Xerox PARC.

The C++ Cond Class

Condition variables are similar to monitors in that they allow a thread to enter a critical region, test a condition, and sleep inside the critical region while releasing its lock. Another thread then is free to enter the critical region, change the condition, and eventually signal the sleeping thread, which resumes at the point where it went to sleep and with the critical region once again locked.

Note that condition variables provide a subset of the functionality of monitors, so a monitor can always be used instead of a condition variable. However, condition variables are smaller, which may be important if you are seriously constrained with respect to memory.

Condition variables are provided by the `IceUtil::Cond` class. Here is its interface:

C++

```
class Cond : private noncopyable
{
public:

    Cond();
    ~Cond();

    void signal();
    void broadcast();

    template<typename Lock>
    void wait(const Lock& lock) const;

    template<typename Lock>
    bool timedWait(const Lock& lock, const Time& timeout) const;
};
```

Using a condition variable is very similar to using a monitor. The main difference in the `Cond` interface is that the `wait` and `timedWait` me mber functions are template functions, instead of the entire class being a template. The member functions behave as follows:

- `wait`
  This function suspends the calling thread and, at the same time, releases the lock of the condition variable. A thread suspended inside a call to `wait` can be woken up by another thread that calls `signal` or `broadcast`. When `wait` completes, the suspended thread resumes execution with the lock held.
- `timedWait`
  This function suspends the calling thread for up to the specified timeout. If another thread calls `signal` or `broadcast` and wakes up the suspended thread before the timeout expires, the call returns true and the suspended thread resumes execution with the lock held. Otherwise, if the timeout expires, the function returns false. Wait intervals are represented by instances of the `Time` class.
- `signal`
  This function wakes up a single thread that is currently suspended in a call to `wait` or `timedWait`. If no thread is suspended in a call to `wait` or `timedWait` at the time `signal` is called, the signal is lost (that is, calls to `signal` are *not* remembered if there is no thread to be woken up). Note that signalling does not necessarily run another thread immediately; the thread calling `signal` may continue to run. However, depending on the underlying thread library, `signal` may also cause an immediate context switch to another thread.
- `broadcast`
  This function wakes up all threads that are currently suspended in a call to `wait` or `timedWait`. As for `signal`, calls to `broadcas t` are lost if no threads are suspended at the time.

You must adhere to a few rules for condition variables to work correctly:

- Do not call `wait` or `timedWait` unless you hold the lock.
- When returning from a `wait` call, you *must* re-test the condition before proceeding, just as for a monitor.

In contrast to monitors, which require you to call `notify` and `notifyAll` with the lock held, condition variables permit you to call `signal` a

nd `broadcast` without holding the lock. Here is a code example that changes a condition and signals on a condition variable:

C++

```
Mutex m;
Cond c;

// ...

{
     Mutex::Lock sync(m);

     // Change some condition other threads may be sleeping on...

     c.signal();

     // ...
} // m is unlocked here
```

This code is correct and will work as intended, but it is potentially inefficient. Consider the code executed by the waiting thread:

C++

```
{
    Mutex::Lock sync(m);

    while(!condition)
    {
        c.wait(sync);
    }

    // Condition is now true, do some processing...

} // m is unlocked here
```

Again, this code is correct and will work as intended. However, consider what can happen once the first thread calls `signal`. It is possible that the call to `signal` will cause an immediate context switch to the waiting thread. But, even if the thread implementation does not cause such an immediate context switch, it is possible for the signalling thread to be suspended after it has called `signal`, but before it unlocks the mutex `m`. If this happens, the following sequence of events occurs:

1. The waiting thread is still suspended inside the implementation of `wait` and is now woken up by the call to `signal`.
2. The now-awake thread tries to acquire the mutex `m` but, because the signalling thread has not yet released the mutex, is suspended again waiting for the mutex to be unlocked.
3. The signalling thread is scheduled again and leaves the scope enclosing `sync`, which unlocks the mutex, making the thread waiting for the mutex runnable.
4. The thread waiting for the mutex acquires the mutex and retests its condition.

While the preceding scenario is functionally correct, it is inefficient because it incurs two extra context switches between the signalling thread and the waiting thread. Because context switches are expensive, this can have quite a large impact on run-time performance, especially if the critical region is small and the condition changes frequently.

You can avoid the inefficiency by unlocking the mutex *before* calling signal:

C++

```
Mutex m;
Cond c;

// ...

{
     Mutex::Lock sync(m);

     // Change some condition other threads may be sleeping on...

} // m is unlocked here

c.signal(); // Signal with the lock available
```

By arranging the code as shown, you avoid the additional context switches because, when the waiting thread is woken up by the call to `signal`, it succeeds in acquiring the mutex before returning from `wait` without being suspended and woken up again first.

As for monitors, you should exercise caution in using `broadcast`, particularly if you have many threads waiting on a condition. Condition variables suffer from the same potential problem as monitors with respect to `broadcast`, namely, that all threads that are currently suspended inside `wait` can immediately attempt to acquire the mutex, but only one of them can succeed and all other threads are suspended again. If your application is sensitive to this condition, you may want to consider waking threads in a more controlled manner.

See Also

- The C++ Monitor Class
- The C++ Time Class

The C++ Thread Classes

The server-side Ice run time by default creates a thread pool for you and automatically dispatches each incoming request in its own thread. As a result, you usually only need to worry about synchronization among threads to protect critical regions when you implement a server. However, you may wish to create threads of your own. For example, you might need a dedicated thread that responds to input from a user interface. And, if you have complex and long-running operations that can exploit parallelism, you might wish to use multiple threads for the implementation of that operation.

Ice provides a simple thread abstraction that permits you to write portable source code regardless of the native threading platform. This shields you from the native underlying thread APIs and guarantees uniform semantics regardless of your deployment platform.

On this page:

The C++ `Thread` Class

The basic thread abstraction in Ice is provided by two classes, `ThreadControl` and `Thread` (defined in `IceUtil/Thread.h`):

C++

```
namespace IceUtil
{
    class Time;

    class ThreadControl
    {
    public:
#ifdef _WIN32
        typedef DWORD ID;
#else
        typedef pthread_t ID;
#endif

        ThreadControl();
#ifdef _WIN32
        ThreadControl(HANDLE, DWORD);
#else
        ThreadControl(explicit pthread_t);
#endif
        ID id() const;

        void join();
        void detach();

        static void sleep(const Time&);
        static void yield();

        bool operator==(const ThreadControl&) const;
        bool operator!=(const ThreadControl&) const;

    };

    class Thread :virtual public Shared
    {
    public:
        virtual void run() = 0;

        ThreadControl start(size_t stBytes = 0);
        ThreadControl start(size_t stBytes, int priority);
        ThreadControl getThreadControl() const;
        bool isAlive() const;

        bool operator==(const Thread&) const;
        bool operator!=(const Thread&) const;
        bool operator<(const Thread&) const;
    };
    typedef Handle<Thread> ThreadPtr;
}
```

The `Thread` class is an abstract base class with a pure virtual `run` method. To create a thread, you must specialize the `Thread` class and

implement the `run` method (which becomes the starting stack frame for the new thread). Note that you must not allow any exceptions to escape from `run`. The Ice run time installs an exception handler that calls `::std::terminate` if `run` terminates with an exception.

The remaining member functions behave as follows:

- `start(size_t stBytes = 0)`
  `start(size_t stBytes, int priority)`
  This member function starts a newly-created thread (that is, calls the `run` method). The `stBytes` parameter specifies a stack size (in bytes) for the thread. The default value of zero creates the thread with a default stack size that is determined by the operating system.

  You can also specify a priority for the thread. (If you do not supply a priority, the thread is created with the system default priority.) The priority value is system-dependent; on POSIX systems, the value must be a legal value for the `SCHED_RR` real-time scheduling policy. (`SCHED_RR` requires root privileges.) On Windows systems, the priority value is passed through to the Windows `setThreadPriority` function. Priority Inversion in C++ provides information about how you can deal with priority inversion.

  The return value is a `ThreadControl` object for the new thread.

  You can start a thread only once; calling `start` on an already-started thread raises `ThreadStartedException`.

  If the calling thread passes an invalid priority or, on POSIX systems, does not have root privileges, `start` raises `ThreadSyscallException`.

- `getThreadControl`
  This member function returns a `ThreadControl` object for the thread on which it is invoked. Calling this method before calling `start` raises a `ThreadNotStartedException`.

- `id`
  This method returns the underlying thread ID (`DWORD` for Windows and `pthread_t` for POSIX threads). This method is provided mainly for debugging purposes. Note also that `pthread_t` is, strictly-speaking, an opaque type, so you should not make any assumptions about what you can do with a thread ID.

- `isAlive`
  This method returns false before a thread's `start` method has been called and after a thread's `run` method has completed; otherwise, while the thread is still running, it returns true. `isAlive` is useful to implement a non-blocking join:

  C++

  ```
  ThreadPtr p = new MyThread();
  // ...
  while(p->isAlive())
  {
      // Do something else...
  }
  p->getThreadControl().join(); // Will not block
  ```

- `operator==`
  `operator!=`
  `operator<`
  These member functions compare the in-memory address of two threads. They are provided so you can use `Thread` objects in sorted STL containers.

Note that `IceUtil` also defines the type `ThreadPtr`. This is the usual reference-counted smart pointer to guarantee automatic clean-up: the `Thread` destructor calls `delete this` once its reference count drops to zero.

Implementing Threads in C++

To illustrate how to implement threads, consider the following code fragment:

C++

```
#include <IceUtil/IceUtil.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread
{
public:
    virtual void run()
     {
        for(int i = 0; i < 100; ++i)
        {
            cout << q.get() << endl;
        }
    }
};

class WriterThread : public IceUtil::Thread
{
public:
    virtual void run()
    {
        for(int i = 0; i < 100; ++i)
        {
            q.put(i);
        }
    }
};
```

This code fragment defines two classes, `ReaderThread` and `WriterThread`, that inherit from `IceUtil::Thread`. Each class implements the pure virtual `run` method it inherits from its base class. For this simple example, a writer thread places the numbers from 1 to 100 into an instance of the thread-safe `Queue` class we defined in our discussion of monitors, and a reader thread retrieves 100 numbers from the queue and prints them to `stdout`.

Creating Threads in C++

To create a new thread, we simply instantiate the thread and call its `start` method:

C++

```
IceUtil::ThreadPtr t = new ReaderThread;
t->start();
// ...
```

Note that we assign the return value from `new` to a smart pointer of type `ThreadPtr`. This ensures that we do not suffer a memory leak:

1. When the thread is created, its reference count is set to zero.
2. Prior to calling `run` (which is called by the `start` method), `start` increments the reference count of the thread to 1.
3. For each `ThreadPtr` for the thread, the reference count of the thread is incremented by 1, and for each `ThreadPtr` that is

destroyed, the reference count is decremented by 1.
4. When `run` completes, `start` decrements the reference count again and then checks its value: if the value is zero at this point, the `Thread` object deallocates itself by calling `delete this`; if the value is non-zero at this point, there are other smart pointers that reference this `Thread` object and deletion happens when the last smart pointer goes out of scope.

Note that, for all this to work, you *must* allocate your `Thread` objects on the heap — stack-allocated `Thread` objects will result in deallocation errors:

C++

```
ReaderThread thread;
IceUtil::ThreadPtr t = &thread; // Bad news!!!
```

This is wrong because the destructor of `t` will eventually call `delete`, which has undefined behavior for a stack-allocated object.

Similarly, you *must* use a `ThreadPtr` for an allocated thread. Do not attempt to explicitly delete a thread:

C++

```
Thread* t = new ReaderThread();

// ...

delete t; // Disaster!
```

This will result in a double deallocation of the thread because the thread's destructor will call `delete this`.

It is legal for a thread to call `start` on itself from within its own constructor. However, if so, the thread must not be (very) short lived:

C++

```
class ActiveObject : public Thread
{
public:
    ActiveObject()
    {
        start();
    }

    void done()
    {
        getThreadControl().join();
    }

    virtual void run()
    {
        // *Very* short lived...
    }
};
typedef Handle<ActiveObject> ActiveObjectPtr;

// ...

ActiveObjectPtr ao = new ActiveObject;
```

With this code, it is possible for `run` to complete before the assignment to the smart pointer `ao` completes; in that case, `start` will call `delete this;` before it returns and `ao` ends up deleting an already-deleted object. However, note that this problem can arise only if `run` is indeed very short-lived and moreover, the scheduler allows the newly-created thread to run to completion before the assignment of the return value of `operator new` to `ao` takes place. This is highly unlikely to happen — if you are concerned about this scenario, do not call `start` from within a thread's own constructor. That way, the smart pointer is assigned first, and the thread started second, so the problem cannot arise.

The C++ `ThreadControl` Class

The `start` method returns an object of type `ThreadControl`. The member functions of `ThreadControl` behave as follows:

- `ThreadControl`
  The default constructor returns a `ThreadControl` object that refers to the calling thread. This allows you to get a handle to the current (calling) thread even if you had not previously saved a handle to that thread. For example:

  C++

  ```
  IceUtil::ThreadControl self;    // Get handle to self
  cout << self.id() << endl;      // Print thread ID
  ```

  This example also explains why we have two classes, `Thread` and `ThreadControl`: without a separate `ThreadControl`, it would not be possible to obtain a handle to an arbitrary thread. (Note that this code works even if the calling thread was not created by the Ice run time; for example, you can create a `ThreadControl` object for a thread that was created by the operating system.)

  The (implicit) copy constructor and assignment operator create a `ThreadControl` object that refers to the same underlying thread as the source `ThreadControl` object.

Note that the constructor is overloaded. For Windows, the signature is

C++

```
ThreadControl(HANDLE, DWORD);
```

For Unix, the signature is

C++

```
ThreadControl(pthread_t);
```

These constructors allow you to create a `ThreadControl` object for the specified thread.

- `join`
  This method suspends the calling thread until the thread on which `join` is called has terminated. For example:

  C++

  ```
  IceUtil::ThreadPtr t = new ReaderThread; // Create a thread
  IceUtil::ThreadControl tc = t->start();  // Start it
  tc.join();                               // Wait for it
  ```

  If the reader thread has finished by the time the creating thread calls `join`, the call to `join` returns immediately; otherwise, the creating thread is suspended until the reader thread terminates.

  Note that the `join` method of a thread must be called from only one other thread, that is, only one thread can wait for another thread to terminate. Calling `join` on a thread from more than one other thread has undefined behavior.

  Calling `join` on a thread that was previously joined with or calling `join` on a detached thread has undefined behavior.
  You must join with each thread you create; failure to join with a thread has undefined behavior.

- `detach`
  This method detaches a thread. Once a thread is detached, it cannot be joined with.

  Calling `detach` on an already detached thread, or calling `detach` on a thread that was previously joined with has undefined behavior.

  Note that, if you have detached a thread, you must ensure that the detached thread has terminated before your program leaves its `main` function. This means that, because detached threads cannot be joined with, they must have a life time that is shorter than that of the main thread.

- `sleep`
  This method suspends the calling thread for the amount of time specified by the `Time` class.

- `yield`
  This method causes the calling thread to relinquish the CPU, allowing another thread to run.

- `operator==`
  `operator!=`
  These operators compare thread IDs. (Note that `operator<` is not provided because it cannot be implemented portably.) These operators yield meaningful results only for threads that have not been detached or joined with.

C++ Thread Example

Following is a small example that uses the `Queue` class we defined in our discussion of monitors. We create five writer and five reader threads. The writer threads each deposit 100 numbers into the queue, and the reader threads each retrieve 100 numbers and print them to `s`

`tdout`:

C++

```
#include <vector>
#include <IceUtil/IceUtil.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread
{
    virtual void run()
    {
        for(int i = 0; i < 100; ++i)
        {
            cout << q.get() << endl;
        }
    }
};

class WriterThread : public IceUtil::Thread
{
    virtual void run()
    {
        for(int i = 0; i < 100; ++i)
        {
            q.put(i);
        }
    }
};

int
main()
{
    vector<IceUtil::ThreadControl> threads;
    int i;

    // Create five reader threads and start them
    //
    for(i = 0; i < 5; ++i)
    {
        IceUtil::ThreadPtr t = new ReaderThread;
        threads.push_back(t->start());
    }

    // Create five writer threads and start them
    //
    for(i = 0; i < 5; ++i)
    {
        IceUtil::ThreadPtr t = new WriterThread;
```

```
        threads.push_back(t->start());
    }

    // Wait for all threads to finish
    //
    for(vector<IceUtil::ThreadControl>::iterator i = threads.begin();
i != threads.end(); ++i)
    {
        i->join();
```

```
        }
    }
```

The code uses the `threads` variable, of type `vector<IceUtil::ThreadControl>`, to keep track of the created threads. The code creates five reader and five writer threads, storing the `ThreadControl` object for each thread in the `threads` vector. Once all the threads are created and running, the code joins with each thread before returning from `main`.

Note that you must not leave `main` without first joining with the threads you have created: many threading libraries crash if you return from `main` with other threads still running. (This is also the reason why you must not terminate a program without first calling `Communicator::destroy`; the `destroy` implementation joins with all outstanding threads before it returns.)

See Also

- Smart Pointers for Classes
- The C++ Monitor Class
- The Ice Threading Model
- The C++ Time Class

Priority Inversion in C++

In real-time systems, if you have threads with different priorities, it is possible to encounter a priority inversion. A priority inversion occurs when a low-priority thread prevents a higher-priority thread from running. This situation arises when a low-priority thread acquires a mutex and is pre-empted by one or more medium-priority threads that do not relinquish the CPU. If a high-priority thread then attempts to acquire the mutex locked by the low-priority thread, it will wait (potentially forever) for the medium-priority threads to complete.

One way to deal with this problem is to use a priority inheritance protocol. If a low-priority thread holds a mutex and a high-priority thread attempts to acquire the mutex, the priority of the thread holding the mutex is temporarily raised to the level of the thread waiting for the mutex. This allows the low-priority thread to keep running until it releases the mutex; as soon as it does, its priority is reduced back to its previous level and the high-priority thread acquires the mutex.

Ice supports the priority inheritance protocol on many POSIX platforms. (Windows does not provide such a protocol.)

For POSIX platforms that support the priority inheritance protocol, mutexes by default do not use it. You can use the `getDefaultMutexProtocol` function to retrieve the current default for your platform:

C++

```
namespace IceUtil
{
    enum MutexProtocol { PrioInherit, PrioNone };

    MutexProtocol getDefaultMutexProtocol();
}
```

On POSIX systems that do not support priority inheritance and on Windows, this function always returns `PrioNone`.

The return value of `getDefaultMutexProtocol` determines whether a default-constructed `Mutex` or `RecMutex` uses priority inheritance. By default, this function returns `PrioNone`. You can override this default by explicitly specifying a different protocol when you construct a `Mutex` or `RecMutex`. On Windows, if you specify `PrioInherit` when you construct a mutex, the setting is ignored and the mutex is constructed as if you had specified `PrioNone`.

To change the value returned by `getDefaultMutexProtocol`, you can edit `cpp/config/Make.rules`, modify the value of the `DEFAULT_MUTEX_PROTOCOL` macro, and then rebuild the `Ice` library.

See Also

- The C++ Mutex Class
- The C++ RecMutex Class

The C++ Handle Template

`IceUtil::Handle` implements a smart reference-counted pointer type. Smart pointers are used to guarantee automatic deletion of heap-allocated class instances.

`Handle` is a template class with the following interface:

C++

```
template<typename T>
class Handle : /* ... */
{
public:

    typedef T element_type;

    T* _ptr;

    T* operator->() const;
    T& operator*() const;
    T* get() const;

    operator bool() const;

    void swap(HandleBase& other);

    Handle(T* p = 0);

    template<typename Y>
    Handle(const Handle<Y>& r);

    Handle(const Handle& r);

    ~Handle();

    Handle& operator=(T* p);

    template<typename Y>
    Handle& operator=(const Handle<Y>& r);

    Handle& operator=(const Handle& r);

    template<class Y>
    static Handle dynamicCast(const HandleBase<Y>& r);

    template<class Y>
    static Handle dynamicCast(Y* p);
};

template<typename T, typename U>
bool operator==(const Handle<T>& lhs, const Handle<U>& rhs);
```

```
template<typename T, typename U>
bool operator!=(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator<(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator<=(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator>(const Handle<T>& lhs, const Handle<U>& rhs);
```

```
template<typename T, typename U>
bool operator>=(const Handle<T>& lhs, const Handle<U>& rhs);
```

Note that the actual implementation is split into a base and a derived class. For simplicity, we show the combined interface here. If you want to see the full implementation detail, it can be found in `IceUtil/Handle.h`.

The template argument must be a class that derives from `Shared` or `SimpleShared` (or that implements reference counting with the same interface as these classes).

This is quite a large interface, but all it really does is to faithfully mimic the behavior of ordinary C++ class instance pointers. Rather than discussing each member function in detail, we provide a simple overview here that outlines the most important points. Please see the discussion of Ice objects for more examples of using smart pointers.

`element_type`

This type definition follows the STL convention of defining the element type with the fixed name `element_type` so you can use it for template programming or the definition of generic containers.

`_ptr`

This data member stores the pointer to the underlying heap-allocated class instance.

Constructors, copy constructor, and assignment operators

These member functions allow you to construct, copy, and assign smart pointers as if they were ordinary pointers. In particular, the constructor and assignment operator are overloaded to work with raw C++ class instance pointers, which results in the "adoption" of the raw pointer by the smart pointer. For example, the following code works correctly and does not cause a memory leak:

C++

```
typedef Handle<MyClass> MyClassPtr;

void foo(const MyClassPtr&);

// ...

foo(new MyClass); // OK, no leak here.
```

`operator->`, `operator*`, and `get`

The arrow and indirection operators allow you to apply the usual pointer syntax to smart pointers to use the target of a smart pointer. The `get` member function returns the class instance pointer to the underlying reference-counted class instance; the return value is the value of `_ptr`.

`dynamicCast`

This member function works exactly like a C++ `dynamic_cast`: it tests whether the argument supports the specified type and, if so, returns a non-null pointer; if the target does not support the specified type, it returns null.

The reason for not using an actual `dynamic_cast` and using a `dynamicCast` function instead is that `dynamic_cast` only operates on pointer types, but `IceUtil::Handle` is a class.

For example:

C++

```
MyClassPtr p = ...;
MyOtherClassPtr o = ...;

o = MyOtherClassPtr::dynamicCast(p);
if(o)
{
    // o points at an instance of type MyOtherClass.
}
else
{
    // p points at something that is
    // not compatible with MyOtherClass.
}
```

Note that this example also illustrates the use of `operator bool`: when used in a boolean context, a smart pointer returns true if it is non-null and false otherwise.

Comparison operators: `==, !=, <, <=, >, >=`

The comparison operators for smart pointers delegate to the operators of the underlying class, therefore the author of the reference-counted class defines the semantics of smart pointer comparison. For example, in the case of Slice classes, the base class `Ice::Object` implements the comparison operators in terms of pointer addresses. On the other hand, the base class for Slice proxies implements comparison using value semantics.

See Also

- Smart Pointers for Classes
- C++98 Mapping for Interfaces
- The C++ Shared and SimpleShared Classes

The C++ Handle Template Adaptors

`IceUtil` provides adaptors that support use of smart pointers with STL algorithms. Each template function returns a corresponding function object that is for use by an STL algorithm. The adaptors are defined in the header `IceUtil/Functional.h`.

Here is a list of the adaptors:

```
memFun
memFun1
voidMemFun
voidMemFun1

secondMemFun
secondMemFun1
secondVoidMemFun
secondVoidMemFun1

constMemFun
constMemFun1
constVoidMemFun
constVoidMemFun1

secondConstMemFun
secondConstMemFun1
secondConstVoidMemFun
secondConstVoidMemFun1
```

As you can see, the adaptors are in two groups. The first group operates on non-const smart pointers, whereas the second group operates on `const` smart pointers (for example, on smart pointers declared as `const MyClassPtr`).

Each group is further divided into two sub-groups. The adaptors in the first group operate on the target of a smart pointer, whereas the `second<`*`name`*`>` adapters operate on the second element of a pair, where that element is a smart pointer.

Each of the four sub-groups contains four adaptors:
`memFun`

This adaptor is used for member functions that return a value and do not accept an argument. For example:

C++

```
class MyClass : public IceUtil::Shared
{
public:
    MyClass(int i) : _i(i) {}
    int getVal() { return _i; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(42));
mcp.push_back(new MyClass(99));

transform(mcp.begin(), mcp.end(),
          ostream_iterator<int>(cout, " "),
          IceUtil::memFun(&MyClass::getVal));
cout << endl;
```

This code invokes the member function `getVal` on each instance that is pointed at by smart pointers in the vector `mcp` and prints the return value of `getVal` on `cout`, separated by spaces. The output from this code is:

```
42 99
```

`memFun1`

This adaptor is used for member functions that return a value and accept a single argument. For example:

C++

```
class MyClass : public IceUtil::Shared
{
public:
    MyClass(int i) : _i(i) {}
    int plus(int v) { return _i + v; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

int A[3] = { 5, 7, 9 };
transform(mcp.begin(), mcp.end(), A,
          ostream_iterator<int>(cout, " "),
          IceUtil::memFun1(&MyClass::plus));
cout << endl;
```

This code invokes the member function `plus` on each instance that is pointed at by smart pointers in the vector `mcp` and prints the return value of a call to `plus` on `cout`, separated by spaces. The calls to `plus` are successively passed the values stored in the array `A`. The output from this code is:

```
7 11 15
```

`voidMemFun`

This adaptor is used for member functions that do not return a value and do not accept an argument. For example:

C++

```
class MyClass : public IceUtil::Shared
{
public:
    MyClass(int i) : _i(i) {}
    void print() { cout << _i << endl; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

for_each(mcp.begin(), mcp.end(), IceUtil::voidMemFun(&MyClass::print));
```

This code invokes the member function `print` on each instance that is pointed at by smart pointers in the vector `mcp`. The output from this code is:

```
2
4
6
```

`voidMemFun1`

This adaptor is used for member functions that do not return a value and accept a single argument. For example:

C++

```
class MyClass : public IceUtil::Shared
{
public:
    MyClass(int i) : _i(i) {}
    void printPlus(int v) { cout << _i + v << endl; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

for_each(
    mcp.begin(), mcp.end(),
    bind2nd(IceUtil::voidMemFun1(&MyClass::printPlus), 3));
```

This code invokes the member function `printPlus` on each instance that is pointed at by smart pointers in the vector `mcp`. The output from this code is:

```
5
7
9
```

As mentioned earlier, the `second<`*name*`>` versions of the adaptors operate on the second element of a `std::pair<T1, T2>`, where `T2` must be a smart pointer. Most commonly, these adaptors are used to apply an algorithm to each lookup value of a map or multi-map. Here is an example:

C++

```
class MyClass : public IceUtil::Shared
{
public:
    MyClass(int i) : _i(i) {}
    int plus(int v) { return _i + v; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

map<string, MyClassPtr> m;
m["two"] = new MyClass(2);
m["four"] = new MyClass(4);
m["six"] = new MyClass(6);

int A[3] = { 5, 7, 9 };
transform(
    m.begin(), m.end(), A,
    ostream_iterator<int>(cout, " "),
    IceUtil::secondMemFun1<int, string, MyClass>(&MyClass::plus));
```

This code invokes the `plus` member function on the class instance denoted by the `second` smart pointer member of each pair in the dictionary `m`. The output from this code is:

```
9 13 11
```

Note that `secondMemFun1` is a template that requires three arguments: the return type of the member function to be invoked, the key type of the dictionary, and the type of the class that is pointed at by the smart pointer.

In general, the `second<`*name*`>` adaptors require the following template arguments:

C++

```
secondMemFun<R, K, T>
secondMemFun1<R, K, T>
secondVoidMemFun<K, T>
secondVoidMemFun<K, T>
```

where `R` is the return type of the member function, `K` is the type of the first member of the pair, and `T` is the class that contains the member function.

See Also

- The C++ Handle Template

The C++ ScopedArray Template

`IceUtil::ScopedArray` is a smart pointer class similar to `Handle`. However, instead of managing the memory for class instances, `ScopedArray` manages memory for an array. This class is provided mainly for use with the stream API. However, you can use it with arrays for other purposes.

Here is the definition of the template in full:

C++

```
template<typename T>
class ScopedArray : private IceUtil::noncopyable
{
public:
    explicit ScopedArray(T* ptr = 0)
        : _ptr(ptr) { }

    ScopedArray(const ScopedArray& other)
    {
        _ptr = other._ptr;
        const_cast<ScopedArray&>(other)._ptr = 0;
    }

    ~ScopedArray()
    {
        if(_ptr != 0)
        {
            delete[] _ptr;
        }
    }

    void reset(T* ptr = 0)
     {
        assert(ptr == 0 || ptr != _ptr);
        if(_ptr != 0)
        {
            delete[] _ptr;
        }
        _ptr = ptr;
    }

    T& operator[](size_t i) const
    {
        assert(_ptr != 0);
        assert(i >= 0);
        return _ptr[i];
    }

    T* get() const
    {
        return _ptr;
    }
```

```
    void swap(ScopedArray& a)
    {
        T* tmp = a._ptr;
        a._ptr = _ptr;
        _ptr = tmp;
    }

private:
```

```
    T* _ptr;
};
```

The class allows you to allocate an array on the heap and assign its pointer to a `ScopedArray` instance. When the instance goes out of scope, it calls `delete[]` on the array, so you do not need to deallocate the array explicitly yourself. This greatly reduces the risk of a memory leak due to an early return or uncaught exception.

See Also

- C++ Streaming Interfaces

The C++ Shared and SimpleShared Classes

`IceUtil::Shared` and `IceUtil::SimpleShared` are base classes that implement the reference-counting mechanism for smart pointers. The two classes provide identical interfaces; the difference between `Shared` and `SimpleShared` is that `SimpleShared` is not thread-safe and, therefore, can only be used if the corresponding class instances are accessed only by a single thread. (`SimpleShared` is marginally faster than `Shared` because it avoids the locking overhead that is incurred by `Shared`.)

The interface of `Shared` looks as follows. (Because `SimpleShared` has the same interface, we do not show it separately here.)

C++

```
class Shared
{
public:
    Shared();
    Shared(const Shared&);
    virtual ~Shared();

    Shared& operator=(const Shared&);

    virtual void __incRef();
    virtual void __decRef();
    virtual int __getRef() const;
    virtual void __setNoDelete(bool);
};
```

The class maintains a reference that is initialized to zero by the constructor. `__incRef` increments the reference count and `__decRef` decrements it. If, during a call to `__decRef`, after decrementing the reference count, the reference count drops to zero, `__decRef` calls `delete this`, which causes the corresponding class instance to delete itself. The copy constructor increments the reference count of the copied instance, and the assignment operator increments the reference count of the source and decrements the reference count of the target.

The `__getRef` member function returns the value of the reference count and is useful mainly for debugging.

The `__setNoDelete` member function can be used to temporarily disable self-deletion and re-enable it again. This provides exception safety when you initialize a smart pointer with the `this` pointer of a class instance during construction.

To create a class that is reference-counted, you simply derive the class from `Shared` and define a smart pointer type for the class, for example:

C++

```
class MyClass : public IceUtil::Shared
{
    // ...
};

typedef IceUtil::Handle<MyClass> MyClassPtr;
```

See Also

- The C++ Handle Template
- Smart Pointers for Classes

The C++ Time Class

The `Time` class provides basic facilities for getting the current time, constructing time intervals, adding and subtracting times, and comparing times:

C++

```
namespace IceUtil
{

    typedef ... Int64;

    class Time
    {
    public:
        enum Clock { Realtime, Monotonic };
        Time(Clock = Realtime);
        static Time now();
        static Time seconds(Int64);
        static Time milliSeconds(Int64);
        static Time microSeconds(Int64);

        Int64 toSeconds() const;
        Int64 toMilliSeconds() const;
        Int64 toMicroSeconds() const;

        double toSecondsDouble() const;
        double toMilliSecondsDouble() const;
        double toMicroSecondsDouble() const;

        std::string toDateTime() const;
        std::string toDuration() const;

        Time operator-() const;

        Time operator-(const Time&) const;
        Time operator+(const Time&) const;

        Time operator*(int) const;
        Time operator*(Int64) const;
        Time operator*(double) const;

        double operator/(const Time&) const;
        Time operator/(int) const;
        Time operator/(Int64) const;
        Time operator/(double) const;

        Time& operator-=(const Time&);
        Time& operator+=(const Time&);

        Time& operator*=(int);
```

```
        Time& operator*=(Int64);
        Time& operator*=(double);

        Time& operator/=(int);
        Time& operator/=(Int64);
        Time& operator/=(double);

        bool operator<(const Time&) const;
        bool operator<=(const Time&) const;
        bool operator>(const Time&) const;
        bool operator>=(const Time&) const;
        bool operator==(const Time&) const;
        bool operator!=(const Time&) const;

#ifndef _WIN32
        operator timeval() const;
#endif
    };
```

```
    std::ostream& operator<<(std::ostream&, const Time&);
}
```

The member functions behave as follows:

`Time`

Internally, the `Time` class stores ticks in microsecond units. For absolute time, this is the number of microseconds since the Unix epoch (00:00:00 UTC on 1 Jan. 1970). For durations, this is the number of microseconds in the duration. The default constructor initializes the tick count to zero and selects the real-time clock. Constructing `Time` with an argument of `Monotonic` selects the monotonic clock on platforms that support it; the real-time clock is used on other platforms.

`now`

This function constructs a `Time` object that is initialized to the current time of day.

`seconds, milliSeconds, microSeconds`

These functions construct `Time` objects from the argument in the specified units. For example, the following statement creates a time duration of one minute:

C++

```
IceUtil::Time t = IceUtil::Time::seconds(60);
```

`toSeconds, toMilliSeconds, toMicroSeconds`

The member functions provide explicit conversion of a duration to seconds, milliseconds, and microseconds, respectively. The return value is a 64-bit signed integer (`IceUtil::Int64`). For example:

C++

```
IceUtil::Time t = IceUtil::Time::milliSeconds(2000);
IceUtil::Int64 secs = t.toSeconds(); // Returns 2
```

`toSecondsDouble, toMilliSecondsDouble, toMicroSecondsDouble`

The member functions provide explicit conversion of a duration to seconds, milliseconds, and microseconds, respectively. The return value is of type `double`.

`toDateTime`

This function returns a human-readable representation of a `Time` value as a date and time.

`toDuration`

This function returns a human-readable representation of a `Time` value as a duration.

`Operators`

`Time` provides operators that allow you to add, subtract, multiply, and divide times. For example:

C++

```
IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
IceUtil::Time oneMinuteAgo = IceUtil::Time::now() - oneMinute;
```

The multiplication and division operators permit you to multiply and divide a duration. Note that these operators provide overloads for `int`, `long long`, and `double`.

The comparison operators allow you to compare times and time intervals with each other, for example:

```
IceUtil::Time oneMinute  = IceUtil::Time::seconds(60);
IceUtil::Time twoMinutes = IceUtil::Time::seconds(120);
assert(oneMinute < twoMinutes);
```

The `timeval` operator converts a `Time` object to a `struct timeval`, defined as follows:

C++

```
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

The conversion is useful for API calls that require a `struct timeval` argument, such as `select`. To convert a duration into a `timeval` structure, simply assign a `Time` object to a `struct timeval`:

C++

```
IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
struct timeval tv;
tv = t;
```

Note that this member function is not available under Windows.

```
std::ostream& operator<<(std::ostream&, Time&);
```

This operator prints the number of whole seconds since the epoch.

See Also

- The C++ Timer and TimerTask Classes

The C++ Timer and TimerTask Classes

The `Timer` class allows you to schedule some code for once-only or repeated execution after some time interval elapses. The code to be executed resides in a class you derive from `TimerTask`:

C++

```
class Timer;
typedef IceUtil::Handle<Timer> TimerPtr;

class TimerTask : public virtual IceUtil::Shared
{
public:
    virtual ~TimerTask() { }
    virtual void runTimerTask() = 0;
};

typedef IceUtil::Handle<TimerTask> TimerTaskPtr;
```

Your derived class must override the `runTimerTask` member function; the code in this method is executed by the timer. If the code you want to run requires access to some program state, you can pass that state into the constructor of your class or, alternatively, set that state via member functions of your class before scheduling it with a timer.

The `Timer` class invokes the `runTimerTask` member function to run your code. The class has the following definition:

C++

```
class Timer : /* ... */
{
public:
    Timer();
    Timer(int priority);

    void schedule(const TimerTaskPtr& task,
const IceUtil::Time& interval);

    void scheduleRepeated(const TimerTaskPtr& task,
const IceUtil::Time& interval);

    bool cancel(const TimerTaskPtr& task);

    void destroy();
};

typedef IceUtil::Handle<Timer> TimerPtr;
```

Intervals are specified using `Time` objects.

The constructor is overloaded to allow you specify a thread priority. The priority controls the priority of the thread that executes your task.

The `schedule` member function schedules an instance of your timer task for once-only execution after the specified time interval has elapsed. Your code is executed by a separate thread that is created by the `Timer` class. The function throws an `IllegalArgumentExcep`

tion if you invoke it on a destroyed timer.

The scheduleRepeated member function runs your task repeatedly, at the specified time interval. Your code is executed by a separate thread that is created by the Timer class; the same thread is used every time your code runs. The function throws an IllegalArgumentException if you invoke it on a destroyed timer.

If your code throws an exception, the Timer class ignores the exception, that is, for a task that is scheduled to run repeatedly, an exception in the current execution does not cancel the next execution.

If your code takes longer to execute than the time interval you have specified for repeated execution, the second execution is delayed accordingly. For example, if you ask for repeated execution once every five seconds, and your code takes ten seconds to complete, then the second execution of your task starts five seconds after the previous execution finishes, that is, the interval specifies the wait time between successive executions.

A TimerTask instance that has already been scheduled with a Timer instance cannot be scheduled again with the same Timer instance until the task has completed or been canceled.

For a single Timer instance, the execution of all registered tasks is serialized. The wait interval applies on a per-task basis so, if you schedule task A at an interval of five seconds, and task B at an interval of ten seconds, successive runs of task A start no sooner than five seconds after the previous task A has finished, and successive runs of task B start no sooner than ten seconds after the previous task B has finished. If, at the time a task is scheduled to run, another task is still running, the new task's execution is delayed until the previous task has finished.

If you want scheduled tasks to run concurrently, you can create several Timer instances; tasks then execute in as many threads concurrently as there are Timer instances.

The cancel member function removes a task from a timer's schedule. In other words, it stops a task that is scheduled from being executed. If you cancel a task while it is executing, cancel returns immediately and the currently running task is allowed to complete normally; that is, cancel does not wait for any currently running task to complete.

The return value is true if cancel removed the task from the schedule. This is the case if you invoke cancel on a task that is scheduled for repeated execution and this was the first time you cancelled that task; subsequent calls to cancel return false. Calling cancel on a task scheduled for once-only execution always returns false, as does calling cancel on a destroyed timer.

The destroy member function removes all tasks from the timer's schedule. If you call destroy from any thread other than the timer's own execution thread, it joins with the currently executing task (if any), so the function does not return until the current task has completed. If you call destroy from the timer's own execution thread, it instead detaches the timer's execution thread. Calling destroy a second time on the same Timer instance has no effect. Similarly, calling cancel on a destroyed timer has no effect.

Note that you must call destroy on a Timer instance before allowing it to go out of scope; failing to do so causes undefined behavior.

Calls to schedule or scheduleRepeated on a destroyed timer raises an IceUtil::IllegalArgumentException.

See Also

- The C++ Time Class
- The C++ Thread Classes

C-Sharp Mapping

Topics

Initialization in C-Sharp

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `Ice.Communicator` object.

A `Communicator` is a local C# object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application.

You initialize the Ice run time by calling `Ice.Util.initialize`, for example:

C#

```
public static void Main(string[] args)
{
    Ice.Communicator communicator = Ice.Util.initialize(ref args);
    ...
}
```

`Ice.Util.initialize` accepts the argument vector that is passed to `Main` by the operating system. The method scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `Ice.Util.initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your `Main` method, you must call `Communicator.destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, in an Ice server, `destroy` waits for any operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your `Main` method to terminate without calling `destroy` first.

The general shape of our `Main` method becomes:

C#

```
using System;

public class App
{
    public static int Main(string[] args)
    {
        int status = 0;
        Ice.Communicator communicator = null;

        try
        {
            // correct but suboptimal, see below
            communicator = Ice.Util.initialize(ref args);
            // ...
        }
        catch(Exception ex)
        {
            Console.Error.WriteLine(ex);
            status = 1;
        }

        if(communicator != null)
        {
            // correct but suboptimal, see below
            communicator.destroy();
        }
        return status;
    }
}
```

This code is a little bit clunky, as we need to make sure the communicator gets destroyed in all paths, including when an exception is thrown.

Fortunately, the `Ice.Communicator` interface implements `IDisposable`: this allows us to call `initialize` in a `using` statement, which disposes of (destroys) the communicator automatically, without an explicit call to the `destroy` method.

The preferred way to initialize the Ice run time in C# is therefore:

C#

```
using System;

public class App
{
    public static int Main(string[] args)
    {
        try
        {
            using(Ice.Communicator
communicator = Ice.Util.initialize(ref args))
            {
                // ...
            } // communicator is destroyed automatically here
        }
        catch(Exception ex)
        {
            Console.Error.WriteLine(ex);
            return 1;
        }
        return 0;
    }
}
```

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-C-Sharp Mapping

The client-side Slice-to-C# mapping defines how Slice data types are translated to C# types, and how clients invoke operations, pass parameters, and handle errors. Much of the C# mapping is intuitive. For example, by default, Slice sequences map to C# arrays, so there is little you have learn in order to use Slice sequences in C#.

The C# API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the C# mapping, you should need no more than the Slice definition of your application and knowledge of the C# mapping rules. In particular, looking through the generated code in order to discern how to use the C# mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **The `Ice` Namespace**
>
> All of the APIs for the Ice run time are nested in the `Ice` namespace, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` namespace are generated from Slice definitions; other parts of the `Ice` namespace provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` namespace throughout the remainder of the manual.

Topics

C-Sharp Mapping for Identifiers

Slice identifiers map to an identical C# identifier. For example, the Slice identifier `Clock` becomes the C# identifier `Clock`. If a Slice identifier is the same as a C# keyword, the corresponding C# identifier is a *verbatim identifier* (an identifier prefixed with @). For example, the Slice identifier `while` is mapped as `@while`.

> You should try to avoid such identifiers as much as possible.

The Slice-to-C# compiler generates classes that inherit from interfaces or base classes in the .NET framework. These interfaces and classes introduce a number of methods into derived classes. To avoid name clashes between Slice identifiers that happen to be the same as an inherited method, such identifiers are prefixed with `ice_` and suffixed with _ in the generated code. For example, the Slice identifier `Clone` maps to the C# identifier `ice_Clone_` if it would clash with an inherited `Clone`. The complete list of identifiers that are so changed is:

`Clone`	`Equals`	`Finalize`
`GetBaseException`	`GetHashCode`	`GetObjectData`
`GetType`	`MemberwiseClone`	`ReferenceEquals`
`ToString`	`checkedCast`	`uncheckedCast`

Note that Slice identifiers in this list are translated to the corresponding C# identifier only where necessary. For example, structures do not derive from `ICloneable`, so if a Slice structure contains a member named `Clone`, the corresponding C# structure's member is named `Clone` as well. On the other hand, classes do derive from `ICloneable`, so, if a Slice class contains a member named `Clone`, the corresponding C# class's member is named `ice_Clone_`.

Also note that, for the purpose of prefixing, Slice identifiers are case-insensitive, that is, both `Clone` and `clone` are escaped and map to `ice_Clone_` and `ice_clone_`, respectively.

See Also

- Identifiers That Are Keywords
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Exceptions

C-Sharp Mapping for Modules

Slice modules map to C# namespaces with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions. For example:

Slice

```
module M1
{
    // Definitions for M1 here...
    module M2
    {
        // Definitions for M2 here...
    }
}

// ...

module M1    // Reopen M1
{
    // More definitions for M1 here...
}
```

This definition maps to the corresponding C# definitions:

C#

```
namespace M1
{
    namespace M2
    {
        // ...
    }
    // ...
}

// ...

namespace M1    // Reopen M1
{
    // ...
}
```

If a Slice module is reopened, the corresponding C# namespace is reopened as well.

See Also

- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences

C-Sharp Mapping for Built-In Types

The Slice built-in types are mapped to C# types as shown below:

Slice	C#
`bool`	`bool`
`byte`	`byte`
`short`	`short`
`int`	`int`
`long`	`long`
`float`	`float`
`double`	`double`
`string`	`string`

Mapping of Slice built-in types to C#.

See Also

- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Exceptions

C-Sharp Mapping for Enumerations

A Slice enumeration maps to the corresponding enumeration in C#. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

Not surprisingly, the generated C# definition is very similar:

C#

```
public enum Fruit { Apple, Pear, Orange }
```

Suppose we modify the Slice definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The generated C# definition now includes an explicit initializer for every enumerator:

C#

```
public enum Fruit { Apple = 0, Pear = 3, Orange = 4 }
```

See Also

- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Exceptions

C-Sharp Mapping for Structures

Ice for .NET supports two different mappings for Slice structures. By default, Slice structures map to C# structures if they (recursively) contain only value types. If a Slice structure (recursively) contains a string, proxy, class, sequence, or dictionary member, it maps to a C# class. A metadata directive allows you to force the mapping to a C# class for Slice structures that contain only value types.

In addition, for either mapping, you can control whether Slice data members are mapped to fields or to properties.

On this page:

Structure Mapping for Structures in C#

Consider the following structure:

Slice

```
struct Point
{
    double x;
    double y;
}
```

This structure consists of only value types and so, by default, maps to a C# partial structure:

C#

```
public partial struct Point
{
    public double x;
    public double y;

    partial void ice_initialize();
    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
        ice_initialize();
    }

    public override int GetHashCode()
    {
       ...
    }

    public override bool Equals(object other)
    {
       ...
    }

    public static bool operator==(Point lhs, Point rhs)
    {
       ...
    }

    public static bool operator!=(Point lhs, Point rhs)
    {
       ...
    }
}
```

For each data member in the Slice definition, the C# structure contains a corresponding public data member of the same name.

The generated constructor accepts one argument for each structure member, in the order in which they are defined in the Slice definition. This allows you to construct and initialize a structure in a single statement:

C#

```
var p = new Point(5.1, 7.8);
```

The generated constructor calls the `ice_initialize` partial method after initializing the struct data members. You can customize struct initialization by providing your own implementation of `ice_initialize`. Note that C# does not allow a value type to declare a default constructor or to assign default values to data members.

The structure overrides the `GetHashCode` and `Equals` methods to allow you to use it as the key type of a dictionary. (Note that the static

two-argument version of `Equals` is inherited from `System.Object`.) Two structures are equal if (recursively) all their data members are equal. Otherwise, they are not equal. For structures that contain reference types, `Equals` performs a deep comparison; that is, reference types are compared for value equality, not reference equality.

Class Mapping for Structures in C#

The mapping for Slice structures to C# structures provides value semantics. Usually, this is appropriate, but there are situations where you may want to change this:

- If you use structures as members of a collection, each access to an element of the collection incurs the cost of boxing or unboxing. Depending on your situation, the performance penalty may be noticeable.
- On occasion, it is useful to be able to assign null to a structure, for example, to support "not there" semantics (such as when implementing parameters that are conceptually optional).

To allow you to choose the correct performance and functionality trade-off, the Slice-to-C# compiler provides an alternative mapping of structures to classes, for example:

Slice

```
["cs:class"] struct Point
{
    double x;
    double y;
}
```

The `"cs:class"` metadata directive instructs the Slice-to-C# compiler to generate a mapping to a C# partial class for this structure. The generated code is almost identical, except that the keyword `struct` is replaced by the keyword `class` and that the class has a default constructor and inherits from `ICloneable`:

C#

```
public partial class Point : _System.ICloneable
{
    public double x;
    public double y;

    partial void ice_initialize();

    public Point()
    {
        ice_initialize();
    }

    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
        ice_initialize();
    }

    public object Clone()
    {
       ...
    }

    public override int GetHashCode()
    {
       ...
    }

    public override bool Equals(object other)
    {
       ...
    }

    public static bool operator==(Point lhs, Point rhs)
    {
       ...
    }

    public static bool operator!=(Point lhs, Point rhs)
    {
       ...
    }
}
```

Some of the generated marshaling code differs for the class mapping of structures, but this is irrelevant to application code.

The class has a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	Zero
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default C# behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

The class also provides a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement:

C#

```
var p = new Point(5.1, 7.8);
```

All generated constructors call the `ice_initialize` partial method after initializing the data members. You can customize class initialization by providing your own implementation of `ice_initialize`.

The `Clone` method performs a shallow memberwise copy, and the comparison methods have the usual semantics (they perform value comparison).

Note that you can influence the mapping for structures only at the point of definition of a structure, that is, for a particular structure type, you must decide whether you want to use the structure or the class mapping. (You cannot override the structure mapping elsewhere, for example, for individual structure members or operation parameters.)

As we mentioned previously, if a Slice structure (recursively) contains a member of reference type, it is automatically mapped to a C# class. (The compiler behaves as if you had explicitly specified the `"cs:class"` metadata directive for the structure.)

Here is our Employee structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The structure contains two strings, which are reference types, so the Slice-to-C# compiler generates a C# class for this structure:

C#

```
public partial class Employee : _System.ICloneable
{
    public long number;
    public string firstName;
    public string lastName;

    partial void ice_initialize();
    public Employee()
    {
        this.firstName = "";
        this.lastName = "";
        ice_initialize();
    }

    public Employee(long number, string firstName, string lastName)
    {
        this.number = number;
        this.firstName = firstName;
        this.lastName = lastName;
        ice_initialize();
    }

    public object Clone()
    {
       ...
    }

    public override int GetHashCode()
    {
       ...
    }

    public override bool Equals(object other)
    {
       ...
    }

    public static bool operator==(Employee lhs, Employee rhs)
    {
       ...
    }

    public static bool operator!=(Employee lhs, Employee rhs)
    {
       ...
    }
}
```

Property Mapping for Structures in C#

You can instruct the compiler to emit property definitions instead of public data members. For example:

Slice

```
["cs:property"] struct Point
{
    double x;
    double y;
}
```

The `"cs:property"` metadata directive causes the compiler to generate a property for each Slice data member:

C#

```
public partial struct Point
{
    private double x_prop;
    public double x
    {
        get
        {
            return x_prop;
        }
        set
        {
            x_prop = value;
        }
    }

    private double y_prop;
    public double y
    {
        get
        {
            return y_prop;
        }
        set
        {
            y_prop = value;
        }
    }

    // Other methods here...
}
```

Note that the properties are non-virtual because C# structures cannot have virtual properties. However, if you apply the `"cs:property"` directive to a structure that contains a member of reference type, or if you combine the `"cs:property"` and `"cs:class"` directives, the generated properties are virtual. For example:

Slice

```
["cs:property", "cs:class"]
struct Point
{
    double x;
    double y;
}
```

This generates the following code:

C#

```
public partial class Point : System.ICloneable
{
    private double x_prop;
    public virtual double x
    {
        get
        {
            return x_prop;
        }
        set
        {
            x_prop = value;
        }
    }

    private double y_prop;
    public virtual double y
    {
        get
        {
            return y_prop;
        }
        set
        {
            y_prop = value;
        }
    }

    // Other methods here...
}
```

See Also

C-Sharp Mapping for Sequences

Ice for .NET supports several different mappings for sequences. By default, sequences are mapped to arrays. You can use metadata directives to map sequences to a number of alternative types:

- `System.Collections.Generic.List`
- `System.Collections.Generic.LinkedList`
- `System.Collections.Generic.Queue`
- `System.Collections.Generic.Stack`
- User-defined custom types that derive from `System.Collections.Generic.IEnumerable<T>`.

The different mappings allow you to map a sequence to a container type that provides the correct performance trade-off for your application.

On this page:

Array Mapping for Sequences in C#

By default, the Slice-to-C# compiler maps sequences to arrays. Interestingly, no code is generated in this case; you simply define an array of elements to model the Slice sequence. For example:

Slice

```
sequence<Fruit> FruitPlatter;
```

Given this definition, to create a sequence containing an apple and an orange, you could write:

C#

```
Fruit[] fp = { Fruit.Apple, Fruit.Orange };
```

Or, alternatively:

C#

```
Fruit fp[] = new Fruit[2];
fp[0] = Fruit.Apple;
fp[1] = Fruit.Orange;
```

The array mapping for sequences is both simple and efficient, especially for sequences that do not need to provide insertion or deletion other than at the end of the sequence.

Mapping to Predefined Generic Containers for Sequences in C#

With metadata directives, you can change the default mapping for sequences to use generic containers provided by .NET. For example:

Slice

```
["clr:generic:List"] sequence<string> StringSeq;
["clr:generic:LinkedList"] sequence<Fruit> FruitSeq;
["clr:generic:Queue"] sequence<int> IntQueue;
["clr:generic:Stack"] sequence<double> DoubleStack;
```

The "`clr:generic:`*type*" metadata directive causes the `slice2cs` compiler to the map the corresponding sequence to one of the containers in the `System.Collections.Generic` namespace. For example, the `Queue` sequence maps to `System.Collections.Generic.Queue<int>` due to its metadata directive.

The predefined containers allow you to select an appropriate space-performance trade-off, depending on how your application uses a sequence. In addition, if a sequence contains value types, such as `int`, the generic containers do not incur the cost of boxing and unboxing and so are quite efficient. (For example, `System.Collections.Generic.List<int>` performs within a few percentage points of an integer array for insertion and deletion at the end of the sequence, but has the advantage of providing a richer set of operations.)

Generic containers can be used for sequences of any element type except objects. For sequences of objects, only `List` is supported because it provides the functionality required for efficient unmarshaling. Metadata that specifies any other generic type is ignored with a warning:

Slice

```
class MyClass
{
    // ...
}

["clr:generic:List"]
sequence<MyClass> MyClassList; // OK

["clr:generic:LinkedList"]
sequence<MyClass> MyClassLinkedList; // Ignored
```

In this example, sequence type `MyClassList` maps to the generic container `System.Collections.Generic.List<MyClass>`, but sequence type `MyClassLinkedList` uses the default array mapping.

Mapping to Custom Types for Sequences in C#

If the array mapping and the predefined containers are unsuitable for your application (for example, because you may need a priority queue, which does not come with .NET), you can implement your own custom containers and direct `slice2cs` to map sequences to these custom containers. For example:

Slice

```
["clr:generic:MyTypes.PriorityQueue"] sequence<int> Queue;
```

This metadata directive causes the Slice `Queue` sequence to be mapped to the type `MyTypes.PriorityQueue`. You must specify the fully-qualified name of your custom type following the `clr:generic:` prefix. This is because the generated code prepends a `global::` qualifier to the type name you provide; for the preceding example, the generated code refers to your custom type as `global::MyTypes.PriorityQueue<int>`.

Your custom type can have whatever interface you deem appropriate, but it must meet the following requirements:

- The custom type must derive from `System.Collections.Generic.IEnumerable<T>`.
- The custom type must provide a readable `Count` property that returns the number of elements in the collection.
- The custom type must provide an `Add` method that appends an element to the end of the collection.
- If (and only if) the Slice sequence contains elements that are Slice classes, the custom type must provide an indexer that sets the value of an element at a specific index. (Indexes, as usual, start at zero.)

As an example, here is a minimal class (omitting implementation) that meets these criteria:

C#

```
public class PriorityQueue<T> : IEnumerable<T>
{
    public IEnumerator<T> GetEnumerator();

    public int Count
        get;

    public void Add(T elmt);

    public T this[int index] // Needed for class elements only.
        set;

    // Other methods and data members here...
}
```

Multi-Dimensional Sequences in C#

Slice permits you to define sequences of sequences, for example:

Slice

```
enum Fruit { Apple, Orange, Pear }
["clr:generic:List"] sequence<Fruit> FruitPlatter;
["clr:generic:LinkedList"] sequence<FruitPlatter> Cornucopia;
```

If we use these definitions as shown, the type of FruitPlatter in the generated code is:

C#

```
System.Collections.Generic.LinkedList<System.Collections.Generic.List<F
ruit>>
```

Here the outer sequence contains elements of type `List<Fruit>`, as you would expect.

Now let us modify the definition to change the mapping of `FruitPlatter` to an array:

Slice

```
enum Fruit { Apple, Orange, Pear }
sequence<Fruit> FruitPlatter;
["clr:LinkedList"] sequence<FruitPlatter> Cornucopia;
```

With this definition, the type of `Cornucopia` becomes:

C#

```
System.Collections.Generic.LinkedList<Fruit[]>
```

The generated code now no longer mentions the type `FruitPlatter` anywhere and deals with the outer sequence elements as an array of `Fruit` instead.

See Also

- Metadata
- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Exceptions

C-Sharp Mapping for Dictionaries

Ice for .NET supports two mappings for dictionaries. By default, a dictionary maps to `System.Collections.Generic.Dictionary<T>`. You can use a metadata directive to map a dictionary to System.Collections.Generic.SortedDictionary instead.

On this page:

Default Mapping for Dictionaries in C#

Here is the definition of our EmployeeMap once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

By default, the Slice-to-C# compiler maps the dictionary to the following type:

C#

```
System.Collections.Generic.Dictionary<long, Employee>
```

Custom Mapping for Dictionaries in C#

You can use the `"clr:generic:SortedDictionary"` metadata directive to change the default mapping to use a sorted dictionary instead:

Slice

```
["clr:generic:SortedDictionary"]
dictionary<long, Employee> EmployeeMap;
```

With this definition, the type of the dictionary becomes:

C#

```
System.Collections.Generic.SortedDictionary<long, Employee>
```

See Also

- Metadata
- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants

- C-Sharp Mapping for Exceptions

C-Sharp Collection Comparison

The utility class `Ice.CollectionComparer` allows you to compare collections for equality:

C#

```
public class CollectionComparer
{
    public static bool
    Equals(System.Collections.IDictionary d1,
System.Collections.IDictionary d2);

    public static bool
    Equals(System.Collections.ICollection c1,
System.Collections.ICollection c2);

    public static bool
    Equals(System.Collections.IEnumerable c1,
System.Collections.IEnumerable c2);
}
```

Equality of the elements in a collection is determined by calling the elements' `Equals` method.

Two dictionaries are equal if they contain the same number of entries with identical keys and values.

Two collections that derive from `ICollection` or `IEnumerable` are equal if they contain the same number of entries and entries compare equal. Note that order is significant, so corresponding entries must not only be equal but must also appear in the same position.

See Also

- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences
- C-Sharp Mapping for Dictionaries
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Exceptions

C-Sharp Mapping for Constants

Here are the sample constant definitions once more:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

C#

```
public abstract class AppendByDefault
{
    public const bool value = true;
}

public abstract class LowerNibble
{
    public const byte value = 15;
}

public abstract class Advice
{
    public const string value = "Don't Panic!";
}

public abstract class TheAnswer
{
    public const short value = 42;
}

public abstract class PI
{
    public const double value = 3.1416;
}

public enum Fruit { Apple, Pear, Orange }

public abstract class FavoriteFruit
{
    public const Fruit value = Fruit.Pear;
}
```

As you can see, each Slice constant is mapped to a class with the same name as the constant. The class contains a member named `value` that holds the value of the constant.

> The mapping to classes instead of to plain constants is necessary because C# does not permit constant definitions at namespace scope.

Slice string literals that contain non-ASCII characters or universal character names are mapped to C# string literals with universal character names. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

C++

```
public abstract class Egg
{
    public const string value = "\u0153uf";
}

public abstract class Heart
{
    public const string value = "c\u0153ur";
}

public abstract class Banana
{
    public const string value = "\ud83c\udf4c";
}
```

See Also

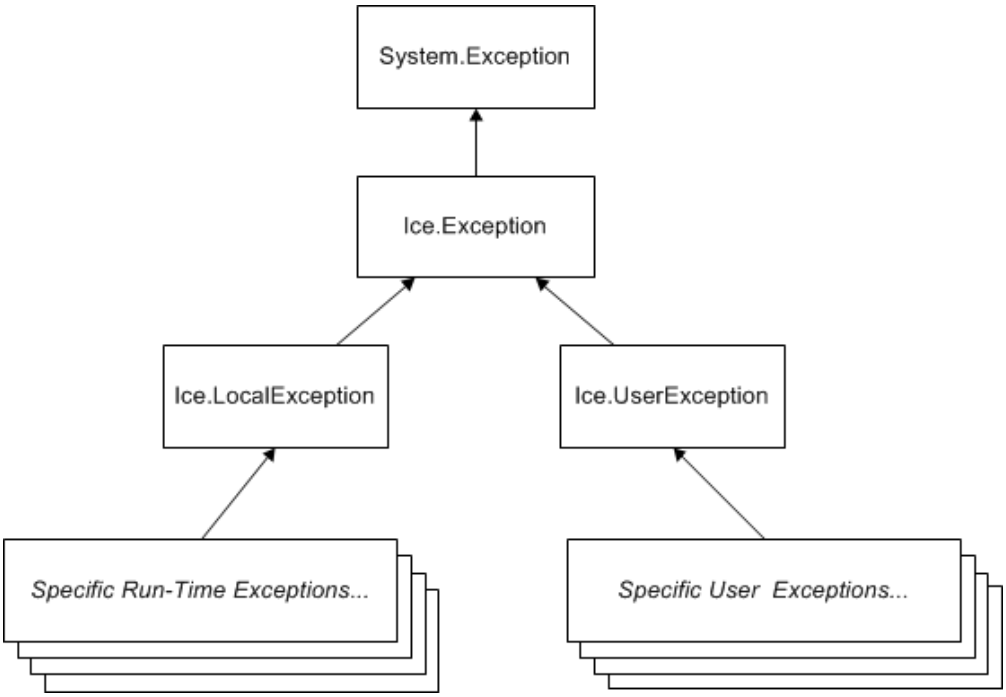
- Constants and Literals
- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Exceptions

C-Sharp Mapping for Exceptions

On this page:

Inheritance Hierarchy for Exceptions in C#

The mapping for exceptions is based on the inheritance hierarchy shown below:



Inheritance structure for exceptions.

The ancestor of all exceptions is `System.Exception`. Derived from that is `Ice.Exception`, which provides the definitions of a number of constructors. `Ice.LocalException` and `Ice.UserException` are derived from `Ice.Exception` and form the base of all run-time and user exceptions, respectively.

The constructors defined in `Ice.Exception` have the following signatures:

C#

```
public abstract class Exception : System.Exception
{
    public Exception();
    public Exception(System.Exception ex);

    public abstract string ice_id();
}
```

Each concrete derived exception class calls these constructors. The second constructor initializes the `InnerException` property of `System.Exception`. (Both constructors set the `Message` property to the empty string.)

The `ice_id` method returns the Slice type id of the exception, for example `"::M::GenericError"`.

C# Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}

exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

C#

```
public partial class GenericError : Ice.UserException
{
    public string reason;

    public GenericError();
    public GenericError(System.Exception ex);
    public GenericError(string reason);
    public GenericError(string reason, System.Exception ex);

    public override string ice_id();

    // GetHashCode and comparison methods defined here,
    // as well as mapping-internal methods.
}

public partial class BadTimeVal : M.GenericError
{
    public BadTimeVal();
    public BadTimeVal(System.Exception ex);
    public BadTimeVal(string reason);
    public BadTimeVal(string reason, System.Exception ex);

    public override string ice_id();

    // GetHashCode and comparison methods defined here,
    // as well as mapping-internal methods.
}

public partial class BadZoneName : M.GenericError
{
    public BadZoneName();
    public BadZoneName(System.Exception ex);
    public BadZoneName(string reason);
    public BadZoneName(string reason, System.Exception ex);

    public override string ice_id();

    // GetHashCode and comparison methods defined here,
    // as well as mapping-internal methods.
}
```

Each Slice exception is mapped to a C# partial class with the same name. For each exception member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.) Optional data members are mapped to instances of the `Ice.Optional` type.

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

All user exceptions are derived from the base class `Ice.UserException`. This allows you to catch all user exceptions generically by

installing a handler for `Ice.UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice.LocalException`, and you can catch all Ice exceptions with a handler for `Ice.Exception`.

All exceptions provide the usual `GetHashCode` and `Equals` methods, as well as the == and != comparison operators.

The generated exception classes also contain other methods that are not shown here; these methods are internal to the C# mapping and are not meant to be called by application code.

C# Default Constructors for User Exceptions

An exception has a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	Zero
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default C# behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

An exception also provides a constructor that accepts one parameter for each data member so that you can construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assign to its members). For a derived exception, this constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order. For each optional data member, the constructor accepts an `Ice.Optional` parameter of the appropriate type.

C# Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice.LocalException` (which, in turn, derives from `Ice.Exception`).

`Ice.LocalException` implements a `Clone` method that is inherited by its derived exceptions, so you can make memberwise shallow copies of exceptions.

By catching exceptions at the appropriate point in the inheritance hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice.Exception`
  This is the root of the inheritance tree for both run-time and user exceptions.
- `Ice.LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `Ice.UserException`
  This is the root of the inheritance tree for user exceptions.
- `Ice.TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `Ice.ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Sequences
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Optional Values
- Versioning
- Object Life Cycle

C-Sharp Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Interfaces in C#

On the client side, a Slice interface maps to a C# interface with member functions that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Slice compiler generates the following definition for use by the client:

C#

```
public interface SimplePrx : Ice.ObjectPrx
{
    void op();
    void op(Ice.OptionalContext context = new Ice.OptionalContext());
}
```

As you can see, the compiler generates a *proxy interface* `SimplePrx`. In general, the generated name is <*interface-name*>`Prx`. If an interface is nested in a module `M`, the generated interface is part of namespace `M`, so the fully-qualified name is `M.`<*interface-name*>`Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a member function of the same name. For the preceding example, we find that the operation `op` has been mapped to the method `op`. The optional parameter `context` is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the `context` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Because all the <*interface-name*>`Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.The proxy references handed out by the Ice run time are always of type <*interface-name*>`Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Interface Inheritance in C#

Inheritance relationships among Slice interfaces are maintained in the generated C# classes. For example:

Slice

```
module M
{
    interface A { ... }
    interface B { ... }
    interface C extends A, B { ... }
}
```

The generated code for `CPrx` reflects the inheritance hierarchy:

C#

```
namespace M
{
    public interface CPrx : APrx, BPrx
    {
        ...
    }
}
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

The `Ice.ObjectPrx` Interface in C#

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

C#

```
namespace Ice
{
    public interface ObjectPrx
    {
        Identity ice_getIdentity();
        bool ice_isA(string id);
        string ice_id();
        void ice_ping();

        int GetHashCode();
        bool Equals(object r);

        // Defined in a helper class:
        //
        public static bool Equals(Ice.ObjectPrx lhs, ObjectPrx rhs);
        public static bool operator==(ObjectPrx lhs, ObjectPrx rhs);
        public static bool operator!=(ObjectPrx lhs, ObjectPrx rhs);

        // ...
    }
}
```

Note that the static methods are not actually defined in `Ice.ObjectPrx`, but in a helper class that becomes a base class of an instantiated proxy. However, this is simply an internal detail of the C# mapping — conceptually, these methods belong with `Ice.ObjectPrx`, so we discuss them here.

The methods behave as follows:

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

  ```
  module Ice
  {
      struct Identity
      {
          string name;
          string category;
      }
  }
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

C#

```
Ice.ObjectPrx o1 = ...;
Ice.ObjectPrx o2 = ...;
Ice.Identity i1 = o1.ice_getIdentity();
Ice.Identity i2 = o2.ice_getIdentity();

if(i1.Equals(i2))
{
    // o1 and o2 denote the same object
}
else
{
    // o1 and o2 denote different objects

}
```

- **ice_isA**
  The ice_isA method determines whether the object denoted by the proxy supports a specific interface. The argument to ice_isA is a type ID. For example, to see whether a proxy of type ObjectPrx denotes a Printer object, we can write:

  C#

  ```
  Ice.ObjectPrx o = ...;
  if(o != null && o.ice_isA("::Printer"))
  {
      // o denotes a Printer object
  }
  else
  {
      // o denotes some other type of object

  }
  ```

  Note that we are testing whether the proxy is null before attempting to invoke the ice_isA method. This avoids getting a NullReferenceException if the proxy is null.

- **ice_ids**
  The ice_ids method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- **ice_id**
  The ice_id method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type BasePrx, with a static type ID of ::Base, the return value of ice_id might be ::Base, or it might something more derived, such as ::Derived.

- **ice_ping**
  The ice_ping method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

- **Equals**
  This method compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the

communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

The `ice_isA`, `ice_ids`, `ice_id`, and `ice_ping` methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Proxy Helpers in C#

For each Slice interface, apart from the proxy interface, the Slice-to-C# compiler creates a helper class: for an interface `Simple`, the name of the generated helper class is `SimplePrxHelper`.

> You can ignore the `ObjectPrxHelperBase` base class — it exists for mapping-internal purposes.

The helper class contains two methods of interest:

C#

```
public class SimplePrxHelper : Ice.ObjectPrxHelperBase, SimplePrx
{
    public static SimplePrx checkedCast(Ice.ObjectPrx b);
    public static SimplePrx checkedCast(
        Ice.ObjectPrx b,
        System.Collections.Generic.Dictionary<string, string> ctx);
    public static SimplePrx uncheckedCast(Ice.ObjectPrx b);
    public static string ice_staticId();

    // ...
}
```

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a non-null reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

C#

```
Ice.ObjectPrx obj = ...;        // Get a proxy from somewhere...

SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if(simple != null)
{
    // Object supports the Simple interface...
}
else
{
    // Object is not of type Simple...

}
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`. (This also explains the need for the helper class: the Ice run time must contact the server, so we cannot use a C# down-cast.)

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

C#

```
Ice.ObjectPrx obj = ...;                    // Get proxy...
ProcessPrx process
= ProcessPrxHelper.uncheckedCast(obj); // No worries...
process.launch(40, 60);                     // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a C# cast, the behavior is undefined.

Another method defined by every helper class is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Using Proxy Methods in C#

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

C#

```
Ice.ObjectPrx proxy = communicator.stringToProxy(...);
proxy = proxy.ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a `checkedCast` or `uncheckedCast` after using a factory method. However, a regular cast is still required, as shown in the example below:

C#

```
Ice.ObjectPrx base = communicator.stringToProxy(...);
HelloPrx hello = HelloPrxHelper.checkedCast(base);
hello = (HelloPrx)hello.ice_invocationTimeout(10000); # Type is
preserved
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in C#

Proxies provide an `Equals` method that compares proxies:

C#

```
public interface ObjectPrx
{
    bool Equals(object r);
}
```

Note that proxy comparison with `Equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `Equals` (or == and `!=`) tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

C#

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if(!p1.Equals(p2))
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `Equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `Equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Ice.Util` class:

C#

```
public sealed class Util
{
    public static int proxyIdentityCompare(ObjectPrx lhs,
ObjectPrx rhs);
    public static int proxyIdentityAndFacetCompare(ObjectPrx lhs,
ObjectPrx rhs);
// ...
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

C#

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if(Ice.Util.proxyIdentityCompare(p1, p2) != 0)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, `-1` if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the

major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

The C# mapping also provides two helper classes in the `Ice` namespace that allow you to insert proxies into hashtables or ordered collections, based on the identity, or the identity plus the facet name:

C#

```
public class ProxyIdentityKey
    : System.Collections.IHashCodeProvider,
      System.Collections.IComparer
{
    public int GetHashCode(object obj);
    public int Compare(object obj1, object obj2);
}

public class ProxyIdentityFacetKey
    : System.Collections.IHashCodeProvider,
      System.Collections.IComparer
{
    public int GetHashCode(object obj);
    public int Compare(object obj1, object obj2);
}
```

Note these classes derive from `IHashCodeProvider` and `IComparer`, so they can be used for both hash tables and ordered collections.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- C-Sharp Mapping for Operations
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

C-Sharp Mapping for Operations

On this page:

Basic C# Mapping for Operations

As we saw in the C# mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

C#

```
NodePrx node = ...;             // Initialize proxy
string name = node.name();      // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

Normal and `idempotent` Operations in C#

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent  string op2();
}
```

The proxy interface for this is:

C#

```
public interface ExamplePrx : Ice.ObjectPrx
{
    string op1();
    string op2();
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

Passing Parameters in C#

In-Parameters in C#

The parameter passing rules for the C# mapping are very simple: parameters are passed either by value (for value types) or by reference (for reference types). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy for these definitions:

C#

```
public interface ClientToServerPrx : Ice.ObjectPrx
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, string[] ss, Dictionary<long, string[]
> st);
    void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

C#

```
ClientToServerPrx p = ...;              // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
bool b = true;
string s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
string[] ss = new string[1];
ss[0] = "Hello world!";
Dictionary<long, string[]> st = new Dictionary<long, string[]>();
st[0] = ss;
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Out-Parameters in C#

Slice `out` parameters simply map to C# `out` parameters.

Here again are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns, out StringSeq ss,
out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Slice compiler generates the following code for these definitions:

C#

```
public interface ServerToClientPrx : Ice.ObjectPrx
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out string[] ss,
             out Dictionary<long, string[]> st);
    void op3(out ServerToClientPrx proxy);
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

C#

```
ClientToServerPrx p = ...;              // Get proxy...

int i;
float f;
bool b;
string s;
p.op1(out i, out f, out b, out s);

NumberAndString ns;
string[] ss;
Dictionary<long, string[]> st;
p.op2(out ns, out ss, out st);

ServerToClientPrx stc;
p.op3(out stc);

System.Console.WriteLine(i);   // Show one of the values
```

Null Parameters in C#

Some Slice types naturally have "empty" or "not there" semantics. Specifically, C# sequences, dictionaries, strings, and structures (if mapped to classes) all can be `null`, but the corresponding Slice types do not have the concept of a null value.

- Slice sequences, dictionaries, and strings cannot be null, but can be empty. To make life with these types easier, whenever you pass a C# `null` reference as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.
- If you pass a C# `null` reference to a Slice structure that is mapped to a C# class as a parameter or return value, the Ice run time automatically sends a structure whose elements are default-initialized. This means that all proxy members are initialized to `null`, sequence and dictionary members are initialized to empty collections, strings are initialized to the empty string, and members that have a value type are initialized to their default values.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are structures, sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid `NullReferenceException`. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Optional Parameters in C#

The mapping for optional parameters is the same as for required parameters, except each optional parameter is encapsulated in an `Ice.Optional` value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

The C# mapping for this operation is shown below:

C#

```
Ice.Optional<int> execute(Ice.Optional<string> params, out
Ice.Optional<float> value, ...);
```

The constructors and conversion operators provided by the `Ice.Optional` type simplify the use of optional parameters:

C#

```
Ice.Optional<int> i;
Ice.Optional<float> v;

i = proxy.execute("--file log.txt", out v); // string converted to
Optional<string>
i = proxy.execute(Ice.Util.None, out v);    // params is unset

if(v.HasValue)
{
    Console.WriteLine("value = " + v.Value);

}
```

A well-behaved program must not assume that an optional parameter always has a value.

Exception Handling in C#

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as C# exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

C#

```
ChildPrx child = ...;   // Get child proxy...

try
{
    child.askToCleanUp();
}
catch(Tantrum t)
{
    System.Console.Write("The child says: ");
    System.Console.WriteLine(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

C#

```
public class Client
{
    static void Main(string[] args)
    {
        try
        {
            ChildPrx child = ...;       // Get child proxy...
            try
            {
                child.askToCleanUp();
                child.praise();         // Give positive feedback...
            }
            catch(Tantrum t)
            {
                System.Console.Write("The child says: ");
                System.Console.WriteLine(t.reason);
                child.scold();          // Recover from error...
            }
        }
        catch(Ice.LocalException e)
        {
            System.Console.WriteLine(e);
        }
    }
}
```

Note that the `ToString` method of exceptions prints the name of the exception, any inner exceptions, and the stack trace. Of course, you can be more selective in the way exceptions are displayed. For example, `e.GetType().Name` returns the (unscoped) name of an exception.

Exceptions and Out-Parameters in C#

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

> This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

See Also

- Operations
- C-Sharp Mapping for Exceptions
- C-Sharp Mapping for Interfaces
- Collocated Invocation and Dispatch

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.

C-Sharp Mapping for Classes

On this page:

Basic C# Mapping for Classes

A Slice class is mapped to a C# class with the same name. By default, the generated class contains a public data member for each Slice data member (just as for structures and exceptions). Alternatively, you can use the property mapping by specifying the `"clr:property"` metadata directive, which generates classes with virtual properties instead of data members.

Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

The Slice compiler generates the following code for this definition:

C#

```
public partial class TimeOfDay : Ice.Value
{
    public short hour;
    public short minute;
    public short second;
    public string tz;

    partial void ice_initialize();

    public TimeOfDay()
    {
        ...
        ice_initialize();
    }

    public TimeOfDay(short hour, short minute, short second, string tz)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
        this.tz = tz;
        ice_initialize();
    }

    public static new string ice_staticId() { ... }
    public override string ice_id() { ... }
}
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice.Value`. This means that all classes implicitly inherit from `Value`, which is the ultimate ancestor of all classes.
2. The generated class contains a public member for each Slice data member.
3. The generated class has a constructor that takes one argument for each data member, as well as a default constructor.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice.Value` in C#

Classes implicitly inherit from a common base class, `Ice.Value`. `Value` is a very simple base class with just a few methods:

C#

```
namespace Ice
{
    [Serializable]
    public abstract class Value : ICloneable
    {
        public static string ice_staticId() { ... }
        public virtual string ice_id() { ... }
        public virtual void ice_preMarshal() {}
        public virtual void ice_postUnmarshal() {}
  public virtual void ice_getSlicedData() {}
        public object Clone() { ... }
    }
}
```

The `Value` methods behave as follows:

- `ice_id`
  This function returns the actual run-time type ID for a class. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_preMarshal`
  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.
- `Clone`
  This method returns a shallow member-wise copy of the value.

Note that the generated class does *not* override `GetHashCode` and `Equals`. This means that classes are compared using shallow reference equality, not value equality (as is used for structures).

Class Data Members in C#

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member. Optional data members are mapped to instances of the `Ice.Optional` type .

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    ["protected"] string tz;    // e.g. GMT, PST, EDT...
}
```

The Slice compiler produces the following generated code for this definition:

C#

```
public partial class TimeOfDay : ...
{
    protected short hour;
    protected short minute;
    protected short second;
    protected string tz;

    partial void ice_initialize();
    public TimeOfDay()
    {
        ice_initialize();
    }

    public TimeOfDay(short hour, short minute, short second, string tz)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
        this.tz = tz;
        ice_initialize();
    }

    // ...
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;   // 0 - 23
    short minute; // 0 - 59
    short second; // 0 - 59
    string tz;    // e.g. GMT, PST, EDT...
}
```

If a protected data member also has the `clr:property` directive, the generated property has protected visibility. Consider the `TimeOfDay` class once again:

Slice

```
["protected", "clr:property"] class TimeOfDay
{
    short hour;   // 0 - 23
    short minute; // 0 - 59
    short second; // 0 - 59
    string tz;    // e.g. GMT, PST, EDT...
}
```

The effects of combining these two metadata directives are shown in the generated code below:

C#

```
public partial class TimeOfDay : ...
{
    private short hour_prop;
    protected short hour
    {
        get
        {
            return hour_prop;
        }
        set
        {
            hour_prop = value;
        }
    }

    // ...
}
```

Refer to the structure mapping for more information on the property mapping for data members.

Class Operations in C#

> Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

Operations in classes are not mapped at all into the corresponding C# class. The generated C# class is the same whether the Slice class has operations or not.

For a class that defines or inherits operations, the Slice-to-C# compiler generates instead a separate <*class-name*>`Disp_` class that can be used to implement an Ice object with these operations. Let's change our example to add a class operation:

Slice

```
class FormattedTimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
    string format();
}
```

The Slice compiler generates the following code:

C#

```
public partial class FormattedTimeOfDay : Ice.Value
{
    // ... operation format() not mapped at all here
}

public interface FormattedTimeOfDayOperations_
{
    string format(Ice.Current current = null);
}

// Disp class for servant implementation
public abstract class FormattedTimeOfDayDisp_ : Ice.ObjectImpl,
FormattedTimeOfDayOperations_
{
    public abstract string format(Ice.Current current = null);
    ...
}
```

This `Disp` class is the *skeleton class* for this Slice class. Skeleton classes are described in the Server-Side C-Sharp Mapping for Interfaces.

Value Factories in C#

While value factories are necessary when using classes with operations (a now deprecated feature), value factories may be used for any kind of class and are *not* deprecated.

Value factories allow you to create classes derived from the C# classes generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
interface Time
{
    TimeOfDay get();
}
```

The default behavior of the Ice run time will create and return an instance of the generated `TimeOfDay` class.

If you wish, you can create your own custom derived class, and tell Ice to create and return these instances instead. For example:

C#

```
public class CustomTimeOfDay : TimeOfDay
{
    public string format() { ... prints formatted data members ... }
}
```

You then create and register a value factory for your custom class with your Ice communicator:

C#

```
Communicator communicator = ...;
communicator.getValueFactoryManager().add((string type) => {
        Debug.Assert(type.Equals(TimeOfDay.ice_staticId()));
        return new CustomTimeOfDay();
    },
    TimeOfDay.ice_staticId());
```

Class Constructors in C#

A class has a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	Zero
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null

`class/interface`	Null

The constructor won't explicitly initialize a data member if the default C# behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

A class also provides a constructor that accepts one argument for each member of the class. This allows you to create and initialize a class in a single statement, for example:

C#

```
TimeOfDay tod = new TimeOfDay(14, 45, 00, "PST"); // 2:45pm
```

For a derived class, the constructor requires one argument for every member of the class, including inherited members. For example, consider the the definition from Class Inheritance once more:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}

class DateTime extends TimeOfDay
{
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
}
```

The Slice compiler generates the following constructors for these types:

C#

```
public partial class TimeOfDay : ...
{
    partial void ice_initialize();
    public TimeOfDay()
    {
        ice_initialize();
    }

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
        ice_initialize();
    }

    // ...
}

public partial class DateTime : TimeOfDay
{
    public DateTime() : base() {}

    public DateTime(short hour,
                    short minute,
                    short second,
                    short day,
                    short month,
                    short year) : base(hour, minute, second)
    {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}
```

If you want to instantiate and initialize a `DateTime` instance, you must either use the default constructor or provide values for all of the data members of the instance, including data members of its base class.

For each optional data member of a class, the constructor accepts an `Ice.Optional` parameter of the appropriate type. You can pass the value `Ice.Util.None` as the value of any optional data member to initialize it to an unset condition.

All generated constructors call the `ice_initialize` partial method after initializing the data members. You can customize class initialization by providing your own implementation of `ice_initialize`.

See Also

- Classes

C-Sharp Mapping for Optional Values

On this page:

The `Ice.Optional` Type

The C# mapping uses a generic structure to hold the values of optional data members and parameters:

C#

```
namespace Ice
{
    public struct NoneType {}

    [Serializable]
    public struct Optional<T> : System.ISerializable
    {
        public Optional(NoneType none);
        public Optional(T v);
        public Optional(Optional<T> v);

        public static explicit operator T(Optional<T> v);
        public static implicit operator Optional<T>(T v);
        public static implicit operator Optional<T>(NoneType v);

        public T Value { get; }
        public bool HasValue { get; }

        public override bool Equals(object other);
        public override int GetHashCode();

        ...
    }
}
```

The `Ice.Optional` type provides constructors and conversion operators that allow you to initialize an instance using the element type or an existing optional value. The default constructor initializes an instance to an unset condition. The `Value` property and conversion operator retrieve the current value held by the instance, or throw `System.InvalidOperationException` if no value is currently set. Use the `HasValue` property to test whether the instance has a value prior to accessing it.

The implicit conversion operators allow you to initialize an `Optional` instance without a cast. However, the conversion operator that extracts the current value is declared as `explicit` and therefore requires a cast:

C#

```
Ice.Optional<int> i = 5; // No cast necessary
int j = i;               // Error!
int k = (int)i;          // OK
```

This operator raises an exception if no value is set, so the proper way to write this is shown below:

C#

```
Ice.Optional<int> i = 5;
int j;
if(i.HasValue)
{
    j = (int)i;
}
else
    ...
```

The `Ice.Util.None` Value

The `Ice.Optional` type provides a constructor and conversion operator that accept `NoneType`. Ice defines an instance of this type, `Ice.Util.None`, that you can use to initialize (or reset) an `Optional` instance to an unset condition:

C#

```
Ice.Optional<int> i = 5;
i = Ice.Util.None;
Debug.Assert(!i.HasValue); // true
```

You can pass `Ice.Util.None` anywhere an `Ice.Optional` value is expected.

See Also

- Optional Data Members

Serializable Objects in C-Sharp

In addition to serializing Slice types, applications may also need to incorporate foreign types into their Slice definitions. Ice allows you to pass CLR serializable objects directly as operation parameters or as fields of another data type. For example:

Slice

```
["clr:serializable:SomeNamespace.CLRClass"]
sequence<byte> CLRObj;
struct MyStruct
{
    int i;
    CLRObj o;
}

interface Example
{
    void op(CLRObj o, MyStruct s);
}
```

The generated code for `MyStruct` contains member `i` of type `int` and a member `o` of type `SomeNamespace.CLRClass`:

C#

```
public partial class MyStruct : _System.ICloneable
{
    public int i;
    public SomeNamespace.CLRClass o;

    // ...
}
```

Similarly, the signature for `op` has parameters of type `CLRClass` and `MyStruct`:

C#

```
void op(SomeNamespace.CLRClass o, MyStruct s);
```

Of course, your client and server code must have an implementation of `CLRClass` that sets the `Serializable` attribute:

C#

```
namespace SomeNamespace
{
    [Serializable]
    public class CLRClass
    {
        // ...
    }
}
```

You can implement this class in any way you see fit — the Ice run time does not place any other requirements on the implementation. However, note that the CLR requires the class to reside in the same assembly for client and server.

See Also

- Serializable Objects

C-Sharp Attribute Metadata Directive

The `slice2cs` compiler supports a metadata directive that allows you to inject C# attribute specifications into the generated code. The metadata directive is `cs:attribute:`. For example:

Slice

```
["cs:attribute:System.Serializable"]
struct Point
{
    double x;
    double y;
}
```

This results in the following code being generated for `S`:

C#

```
[System.Serializable]
public partial struct Point
{
    public double x;
    public double y;
    // ...
}
```

You can apply this metadata directive to any Slice construct, such as structure, operation, or parameter definitions.

You can use this directive also at global level. For example:

Slice

```
[["cs:attribute:assembly: AssemblyDescription(\"My assembly\")"]]
```

This results in the following code being inserted after any `using` directives and before any definitions:

C#

```
[assembly: AssemblyDescription("My assembly")]
```

See Also

- Metadata
- Slice Metadata Directives

Asynchronous Method Invocation (AMI) in C-Sharp

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

Topics

AMI in C-Sharp with Tasks

On this page:

Basic Asynchronous API in C#

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in C#

Besides the synchronous proxy methods, `slice2cs` generates the following asynchronous proxy method:

C#

```
public interface EmployeesPrx : Ice.ObjectPrx
{
    System.Threading.Tasks.Task<string>
    getNameAsync(int number,
                 Ice.OptionalContext context = new
Ice.OptionalContext(),
                 System.IProgress<bool> progress = null,
                 System.Threading.CancellationToken cancel = new
System.Threading.CancellationToken());

    ...
}
```

As you can see, the `getName` operation generates a `getNameAsync` method that accepts several optional parameters:

- a per-invocation context
- a sent callback
- a cancellation token

The `getNameAsync` method sends (or queues) an invocation of `getName`. This method does not block the calling thread. It returns a `Task` t hat you can use in a number of ways, including blocking to obtain the result, configuring a continuation to be executed when the result

becomes available, and polling to check the status of the request.

Here's an example that calls `getNameAsync`:

C#

```
EmployeesPrx e = ...;
Task<string> t = e.getNameAsync(99);

// Continue to do other things here...

string name = t.Result;
```

Because `getNameAsync` does not block, the calling thread can do other things while the operation is in progress.

Asynchronous Mapping for Out Parameters in C#

.NET's standard `Task` API only allows a task to produce one result value. Since a Slice operation could potentially return any number of values, the asynchronous mapping must differ significantly from the synchronous mapping.

The asynchronous mapping depends on how many values an operation returns, including out parameters and a non-`void` return value:

- Zero values
  The corresponding C# method returns an instance of `System.Threading.Tasks.Task`.
- One value
  The corresponding C# method returns an instance of `System.Threading.Tasks.Task<T>` where `T` is the mapped type, regardless of whether the Slice definition of the operation declared it as a return value or as an out parameter. Consider this example:

  Slice

  ```
  interface I
  {
      string op1();
      void op2(out string name);
  }
  ```

  The asynchronous mapping generates corresponding methods with identical signatures:

C#

```
public interface IPrx : Ice.ObjectPrx
{
    System.Threading.Tasks.Task<string>
    op1Async(Ice.OptionalContext context = new
Ice.OptionalContext(),
             System.IProgress<bool> progress = null,
             System.Threading.CancellationToken cancel = new
System.Threading.CancellationToken());

    System.Threading.Tasks.Task<string>
    op2Async(Ice.OptionalContext context = new
Ice.OptionalContext(),
             System.IProgress<bool> progress = null,
             System.Threading.CancellationToken cancel = new
System.Threading.CancellationToken());

    ...
}
```

- Multiple values
  The Slice-to-C# translator generates an extra structure to hold the results of an operation that returns multiple values. This "result type" resides in the same namespace as the proxy interface and has the name *`Interface_Op`*`Result`, where *`Interface`* represents the name of the Slice interface that defines the operation *`Op`*. The leading character of the operation name *`Op`* is always capitalized. The values of out parameters are provided in corresponding data members of the same names. If the operation declares a return value, its value is provided in the data member named `returnValue`. If an out parameter is also named `returnValue`, the data member to hold the operation's return value is named `_returnValue` instead. The result type defines a "one-shot" constructor that accepts and assigns a value for each of its data members. The corresponding C# method returns an instance of `System.Threading.Tasks.Task<T>` where `T` is the result type. Consider this example:

Slice

```
interface Example
{
    double op(int inp1, string inp2, out bool outp1, out long
outp2);
}
```

The generated code looks like this:

C#

```
public struct Example_OpResult
{
    public Example_OpResult(double returnValue, bool outp1, long
outp2) { ... }

    public double returnValue;
    public bool outp1;
    public long outp2;
}

public interface ExamplePrx : Ice.ObjectPrx
{
    System.Threading.Tasks.Task<Example_OpResult>
    opAsync(int inp1, string inp2,
            Ice.OptionalContext context = new
Ice.OptionalContext(),
            System.IProgress<bool> progress = null,
            System.Threading.CancellationToken cancel = new
System.Threading.CancellationToken());

    ...
}
```

Now let's invoke `opAsync` to demonstrate one way of asynchronously executing an action when the invocation completes:

C#

```
ExamplePrx e = ...;
e.opAsync().ContinueWith((t) =>
    {
        try
        {
            var r = t.Result; // Returns Example_OpResult
            Console.WriteLine("returnValue = {0} outp1 = {1} outp2
= {2}", r.returnValue, r.outp1, r.outp2);
        }
        catch (System.AggregateException ex)
        {
            // handle exception...
        }
    });
```

Here's a simpler version that uses the `await` keyword:

C#

```
ExamplePrx e = ...;
try
{
    var r = await e.opAsync();
    Console.WriteLine("returnValue = {0} outp1 = {1} outp2 = {2}",
r.returnValue, r.outp1, r.outp2);
}
catch (Ice.Exception ex)
{
    // handle exception...
}
```

Asynchronous Exception Semantics in C#

If an invocation raises an exception, the exception can be obtained from the task. For example, calling `Wait` on the task raises a `System.AggregateException` whose `InnerException` property contains the actual exception. The task's `Exception` property also returns the `AggregateException` if the exception has already occurred at the time you access the property.

The exception is provided by the task, even if the actual error condition for the exception was encountered during the call to the `opAsync` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that handles the task (instead of being present twice, once where the `opAsync` method is called, and again where the task is handled).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

> Using the `await` keyword to invoke an asynchronous proxy method does not raise `AggregateException` but rather raises the inner exception directly. In other words, the exception semantics with `await` are the same as if you had invoked the synchronous version of the proxy method.

Polling for Completion in C#

The asynchronous API allows you to poll for call completion, which can be useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

C#

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;
while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.Length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

C#

```
using System.Threading;
using System.Threading.Tasks;
...

FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;

var results = new LinkedList<Task>();
const int numRequests = 5;
var sent = new AutoResetEvent(false);
while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    var task = ft.sendAsync(offset, bs, progress:(ss) => sent.Set());
    offset += bs.Length;

    // Wait until this request has been passed to the transport.
    sent.WaitOne();
    results.AddLast(task);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.Count > numRequests)
    {
        var t = results.First;
        results.RemoveFirst();
        t.Wait();
    }
}

// Wait for any remaining requests to complete.
Task.WaitAll(results.ToArray());
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Asynchronous Oneway Invocations in C#

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous method on a oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

The task returned for a oneway invocation completes as soon as the request is successfully written to the client-side transport. The task completes with an exception if an error occurs before the request is successfully written.

Flow Control in C#

Asynchronous method invocations never block the thread that calls the asynchronous proxy method. The Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background.

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport. One of the optional arguments to every asynchronous proxy invocation is a `System.IProgress<bool>`. If you provide a delegate, the Ice run time will eventually invoke it when the request has been sent and provide a boolean argument indicating whether the request was sent synchronously. This argument is true if the entire request could be transferred to the local transport in the caller's thread without blocking, otherwise the argument is false. Furthermore, a value of true indicates that Ice is invoking your delegate recursively from the calling thread, whereas a value of false indicates that Ice is invoking the delegate from an Ice thread pool thread.

Here's a simple example to demonstrate the flow control feature:

C#

```
ExamplePrx proxy = ...;
proxy.doSomethingAsync(progress:(sentSynchronously) =>
    {
        if(sentSynchronously)
        {
            // Entire request was accepted by the transport,
            // called recursively from this thread
        }
        else
        {
            // Request was queued but has now been sent,
            // called from a separate thread
        }
    });
```

Using this feature, you can limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Asynchronous Batch Requests in C#

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

The proxy method `ice_flushBatchRequestsAsync` flushes any batch requests queued by that proxy. In addition, similar methods are available on the communicator and the `Connection` objects. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Canceling Asynchronous Requests in C#

Every asynchronous proxy method accepts an optional instance of the structure `System.Threading.CancellationToken`. The default value is an empty structure, which is equivalent to passing `CancellationToken.None`. If your application requires the ability to cancel an asynchronous request, you need to create a `CancellationTokenSource` from which you can obtain a token. Cancelling a request is achieved by calling `Cancel` on the source object.

> Cancellation prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. Cancellation is a local operation and has no effect on the server. The result of a canceled invocation is an `Ice::InvocationCanceledException`.

Concurrency Semantics for AMI in C#

The default behavior of a call to `ContinueWith` is to execute the continuation in a separate thread from the .NET thread pool. If you're trying to minimize thread context switches, you can pass `TaskContinuationOptions.ExecuteSynchronously` as an additional argument to `ContinueWith`. In this case, the behavior depends on the task's status: if the reply to the proxy invocation has already been received at the time `ContinueWith` is called, the continuation will be invoked by the current thread. If the reply has not yet been received, the continuation will be invoked by an Ice thread pool thread.

> If a dispatcher is configured, the Ice thread pool delegates the execution of the continuation to the dispatcher.

The scheduler that runs continuations can be changed by passing a custom scheduler to `ContinueWith`. The Ice thread pool can be used as a task scheduler, and you can obtain this scheduler by calling the `ice_scheduler` proxy method and passing it to `ContinueWith`. With the Ice thread pool scheduler, the continuation is queued to be executed by the Ice thread pool. If you pass the option `TaskContinuationOptions.ExecuteSynchronously` to `ContinueWith`, and the reply has been received at the time you call `ContinueWith`, your thread will execute the continuation. Therefore, if you want to ensure the continuation is always executed by an Ice thread pool thread (or indirectly the dispatcher, if one is configured), you need to call `ContinueWith`:

- with *proxy*`.ice_scheduler()` as your task scheduler
- without `TaskContinuationOptions.ExecuteSynchronously` as continuation option (or, alternatively, override this option with `TaskContinuationOptions.RunContinuationsAsynchronously)`

When using `async` and `await`, the concurrency semantics are determined by the synchronization context in which you're making the proxy invocation. For example, awaiting an asynchronous proxy invocation from the main thread will invoke the continuation from a .NET thread pool thread. Similarly, awaiting an asynchronous proxy invocation from the GUI thread in a graphical application will invoke the continuation from the GUI thread.

Ice configures a synchronization context for its own thread pool threads, so if you happen to await an asynchronous proxy invocation while in an Ice thread pool thread, the continuation will also be invoked by an Ice thread.

Refer to the flow control discussion for information about the concurrency semantics of the sent callback.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch

AMI in C-Sharp with AsyncResult

> The AMI mapping using the AsyncResult API is deprecated and provided only for backward compatibility. New applications should use the Task API.

On this page:

Basic Asynchronous API in C#

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in C#

Besides the synchronous proxy methods, `slice2cs` generates the following asynchronous proxy methods:

C#

```
public interface EmployeesPrx : Ice.ObjectPrx
{
    Ice.AsyncResult<Demo.Callback_Employees_getName>
    begin_getName(int number);

    Ice.AsyncResult<Demo.Callback_Employees_getName>
    begin_getName(int number,

_System.Collections.Generic.Dictionary<string, string> ctx__);

    string end_getName(Ice.AsyncResult r__);
}
```

Two additional overloads of `begin_getName` are generated for use with generic completion callbacks.

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. (The `begin_` method is overloaded so you can pass a per-invocation context.)

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

C#

```
EmployeesPrx e = ...;
Ice.AsyncResult r = e.begin_getName(99);

// Continue to do other things here...

string name = e.end_getName(r);
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `Ice.AsyncResult`. (The class derives from `System.IAsyncResult`.) This value contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResult` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. Similarly, the `end_` method has one out-parameter for each out-parameter of the corresponding Slice operation (plus the `AsyncResult` parameter). For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

C#

```
Ice.AsyncResult<Demo.Callback_Employees_op>
begin_op(int inp1, string inp2);

double end_op(out bool outp1, out long outp2, Ice.AsyncResult r__);
```

Asynchronous Exception Semantics in C#

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` meth od is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `System.ArgumentException`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

`AsyncResult` Interface in C#

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

C#

```
public interface AsyncResult : System.IAsyncResult
{
    void cancel();

    Ice.Communicator getCommunicator();
    Ice.Connection getConnection();
    ObjectPrx getProxy();
    string getOperation();
    object AsyncState { get; }

    bool IsCompleted { get; }
    void waitForCompleted();

    bool isSent();
    void waitForSent();

    void throwLocalException();

    bool sentSynchronously();

    AsyncResult whenSent(Ice.AsyncCallback cb);
    AsyncResult whenSent(Ice.SentCallback cb);
    AsyncResult whenCompleted(Ice.ExceptionCallback ex);
}

public interface AsyncResult<T> : AsyncResult
{
    AsyncResult<T> whenCompleted(T cb, Ice.ExceptionCallback excb);

    new AsyncResult<T> whenCompleted(Ice.ExceptionCallback excb);
    new AsyncResult<T> whenSent(Ice.SentCallback cb);
}
```

The methods and properties have the following semantics:

- `void cancel()`
  This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. `cancel` is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning `IsCompleted` returns true, and the result of the invocation is an `Ice.InvocationCanceledException`.

- `Communicator getCommunicator()`
  This method returns the communicator that sent the invocation.
- `Connection getConnection()`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `getConnection` method only returns a non-nil value when the `AsyncResult` object is obtained by calling `begin_flushBatchRequests` on a `Connection` object.
- `ObjectPrx getProxy()`
  This method returns the proxy that was used to call the `begin_` method, or nil if the `AsyncResult` object was not obtained via an asynchronous proxy invocation.
- `string getOperation()`
  This method returns the name of the operation.
- `object AsyncState { get; }`
  This property stores an object that you can use to pass shared state from the `begin_` to the `end_` method.
- `bool IsCompleted { get; }`
  This property is true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.
- `void waitForCompleted()`
  This method blocks the caller until the result of an invocation becomes available.
- `bool isSent()`
  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.
- `void waitForSent()`
  This method blocks the calling thread until a request has been written to the client-side transport, or an exception occurs. After `waitForSent` returns, `isSent` returns true if the request was successfully written to the client-side transport, or false if an exception occurred. In the case of a failure, you can call the corresponding `end_` method or `throwLocalException` to obtain the exception.
- `void throwLocalException()`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.
- `bool sentSynchronously()`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).
- `AsyncResult whenSent(Ice.SentCallback cb)`
  `AsyncResult<T> whenSent(Ice.SentCallback cb)`
  `AsyncResult whenCompleted(Ice.ExceptionCallback ex)`
  `AsyncResult<T> whenCompleted(T cb,Ice.ExceptionCallback excb)`
  `AsyncResult<T> whenCompleted(Ice.ExceptionCallback excb)`
  These methods allow you to specify callback methods that are called by the Ice run time. The `whenSent` methods set a callback that triggers when an asynchronous invocation is written to the client-side transport. The `whenCompleted` methods set a callback that triggers when an asynchronous invocation completes (also see Generic Completion Callbacks in C#).

Polling for Completion in C#

The `AsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
};
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

C#

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;
while (!file.eof()) {
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.Length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

C#

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;

LinkedList<Ice.AsyncResult> results = new LinkedList<Ice.AsyncResult>();
const int numRequests = 5;

while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice.AsyncResult r = ft.begin_send(offset, bs);
    offset += bs.Length;

    // Wait until this request has been passed to the transport.
    r.waitForSent();
    results.AddLast(r);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.Count > numRequests)
    {
        Ice.AsyncResult r = results.First;
        results.RemoveFirst();
        r.waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while(results.Count > 0)
{
    Ice.AsyncResult r = results.First;
    results.RemoveFirst();
    r.waitForCompleted();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Generic Completion Callbacks in C#

The `begin_` method is overloaded to allow you to provide completion callbacks. Here are the corresponding methods for the `getName` operation:

C#

```
Ice.AsyncResult begin_getName(
        int number,
        Ice.AsyncCallback cb__,
        object cookie__);

Ice.AsyncResult begin_getName(
        int number,
        _System.Collections.Generic.Dictionary<string, string> ctx__,
        Ice.AsyncCallback cb__,
        object cookie__);
```

The second version of `begin_getName` lets you override the default context. (We discuss the purpose of the `cookie` parameter in the next section.) Following the in-parameters, the `begin_` method accepts a parameter of type `Ice.AsyncCallback`, which is a delegate for a callback method. The Ice run time invokes the callback method when an asynchronous operation completes. Your callback method must have `void` return type and accept a single parameter of type `AsyncResult`, for example:

C#

```
private class MyCallback
{
    public void finished(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try
        {
            string name = e.end_getName(r);
            System.Console.WriteLine("Name is: " + name);
        }
        catch(Ice.Exception ex)
        {
            System.Console.Err.WriteLine("Exception is: " + ex);
        }
    }
}
```

The implementation of your callback method must call the `end_` method. The proxy for the call is available via the `getProxy` method on the `AsyncResult` that is passed by the Ice run time. The return type of `getProxy` is `Ice.ObjectPrx`, so you must down-cast the proxy to its correct type.

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. If you allow an exception to escape from the callback method, the Ice run time produces a log entry by default and ignores the exception. (You can disable the log message by setting the property `Ice.Warn.AMICallback` to zero.)

To inform the Ice run time that you want to receive a callback for the completion of the asynchronous call, you pass a delegate for your

callback method to the `begin_` method:

C#

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
Ice.AsyncCallback del = new Ice.AsyncCallback(cb.finished);

e.begin_getName(99, del, null);
```

The trailing `null` argument specifies a cookie, which we will discuss shortly.

You can avoid explicit instantiation of the delegate and, more tersely, write:

C#

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb.finished, null);
```

In fact, there's often no need to define a callback class at all. An anonymous method is one alternative:

C#

```
EmployeesPrx e = ...;

e.begin_getName(99,
    delegate(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try
        {
            string name = e.end_getName(r);
            System.Console.WriteLine("Name is: " + name);
        }
        catch(Ice.Exception ex)
        {
            System.Console.Err.WriteLine("Exception is: " + ex);
        }
    }, null);
```

A lambda is another alternative:

C#

```
EmployeesPrx e = ...;

e.begin_getName(99,
    r => {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try
        {
            string name = e.end_getName(r);
            System.Console.WriteLine("Name is: " + name);
        }
        catch(Ice.Exception ex)
        {
            System.Console.Err.WriteLine("Exception is: " + ex);
        }
    }, null);
```

Using Cookies for Generic Completion Callbacks in C#

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

The API allows you to pass such state by providing a cookie. A cookie is any class instance; the class can contain whatever data you want to pass, as well as any methods you may want to add to manipulate that data.

Here is an example implementation that stores a `Widget`. (We assume that this class provides whatever methods are needed by the `end_` method to update the display.) When you call the `begin_` method, you pass the appropriate cookie instance to inform the `end_` method how to update the display:

C#

```
// Invoke the getName operation with different widget cookies.
MyCallback cb = ...;
e.begin_getName(99, cb.finished, widget1);
e.begin_getName(24, cb.finished, widget2);
```

The `end_` method can retrieve the cookie from the `AsyncResult` by reading the `AsyncState` property. For this example, we assume that widgets have a `writeString` method that updates the relevant UI element:

C#

```
public void finished(Ice.AsyncResult r)
{
    EmployeesPrx e = (EmployeesPrx)r.getProxy();
    Widget widget = (Widget)r.AsyncState;
    try
    {
        string name = e.end_getName(r);
        widget.writeString(name);
    }
    catch(Ice.Exception ex)
    {
        handleException(ex);
    }
}
```

The cookie provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same cookie instance to multiple invocations.

Type-Safe Completion Callbacks in C#

The generic callback API is not entirely type-safe:

- You must down-cast the return value of `getProxy` to the correct proxy type before you can call the `end_` method.
- You must call the correct `end_` method to match the operation called by the `begin_` method.
- You must remember to catch exceptions when you call the `end_` method; if you forget to do this, you will not know that the operation failed.

`slice2cs` generates an additional type-safe API that takes care of these chores for you. To use type-safe callbacks, you supply delegates for two callback methods:

- a success callback that is called if the operation succeeds
- a failure callback that is called if the operation raises an exception

Here is a callback class for an invocation of the `getName` operation:

C#

```
public class MyCallback
{
    public void getNameCB(string name)
    {
        System.Console.WriteLine("Name is: " + name);
    }

    public void failureCB(Ice.Exception ex)
    {
        System.Console.Err.WriteLine("Exception is: " + ex);
    }
}
```

The callback methods can have any name you prefer and must have `void` return type. The failure callback always has a single parameter of type `Ice.Exception`. The success callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the success callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

At the calling end, you call the `begin_` method as follows:

C#

```
MyCallback cb = new MyCallback();

e.begin_getName(99).whenCompleted(cb.getNameCB, cb.failureCB);
```

Note the `whenCompleted` method on the `AsyncResult` that is returned by the `begin_` method. This method establishes the link between the `begin_` method and the callbacks that are called by the Ice run time by setting the delegates for the success and failure methods.

It is legal to pass a null delegate for the success or failure methods. For the success callback, this is legal only for operations that have `void` return type and no out-parameters. This is useful if you do not care when the operation completes but want to know if the call failed. If you pass a null exception delegate, the Ice run time will ignore any exception that is raised by the invocation.

Defining a callback class as we've shown above is only necessary in practice when the callback has additional state to manage. In many cases, there's no need for a callback class and the delegates can be defined inline:

C#

```
e.begin_getName(99).whenCompleted(
    delegate(string name)
    {
        System.Console.WriteLine("Name is: " + name);
    },
    delegate(Ice.Exception ex)
    {
        System.Console.Err.WriteLine("Exception is: " + ex);
    });
```

Using lambda functions makes the callbacks even more compact:

C#

```
e.begin_getName(99).whenCompleted(
    name => {
        System.Console.WriteLine("Name is: " + name);
    },
    ex => {
        System.Console.Err.WriteLine("Exception is: " + ex);
    });
```

Using Cookies for Type-Safe Completion Callbacks in C#

The type-safe API does not support cookies. If you want to pass state from the `begin_` method to the `end_` method, you must use the generic API or, alternatively, place the state into the callback class containing the callback methods. Here is a simple implementation of a callback class that stores a widget that can be retrieved by the `end_` method:

C#

```
public class MyCallback
{
    public MyCallback(Widget w)
    {
        _w = w;
    }

    private Widget _w;

    public void getNameCB(string name)
    {
        _w.writeString(name);
    }

    public void failureCB(Ice.Exception ex)
    {
        _w.writeError(ex);
    }
}
```

When you call the `begin_` method, you pass the appropriate callback instance to inform the `end_` method how to update the display:

C#

```
EmployeesPrx e = ...;
Widget widget1 = ...;
Widget widget2 = ...;

MyCallback cb1 = new MyCallback(widget1);
MyCallback cb2 = new MyCallback(widget2);

// Invoke the getName operation with different widget callbacks.

e.begin_getName(99).whenCompleted(cb1.getNameCB, cb1.failureCB);
e.begin_getName(24).whenCompleted(cb2.getNameCB, cb2.failureCB);
```

Asynchronous Oneway Invocations in C#

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws a `System.ArgumentException`.

For the generic API, the callback method looks exactly as for a twoway invocation. However, for oneway invocations, the Ice run time does not call the callback method unless the invocation raised an exception during the `begin_` method ("on the way out").

For the type-safe API, you only specify a delegate for the failure method. For example, here is how you could call `ice_ping` asynchronously :

C#

```
ObjectPrx p = ...;
MyCallback cb = new MyCallback();
p.begin_ice_ping().whenCompleted(cb.failureCB);
```

Flow Control in C#

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

For the generic API, you can create an additional callback method:

C#

```
public class MyCallback
{
    public void finished(Ice.AsyncResult r)
    {
        // ...
    }

    public void sent(Ice.AsyncResult r)
    {
        // ...
    }
}
```

As with any other callback method, you are free to choose any name you like. For this example, the name of the callback method is `sent`. You inform the Ice run time that you want to be informed when a call has been passed to the local transport by calling `whenSent`:

C#

```
MyCallback cb = new MyCallback();

e.begin_getName(99).whenCompleted(cb.getNameCB,
cb.failureCB).whenSent(cb.sent);
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the `sent` method from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the `sent` method from a different thread once it has written the request to the local transport. In addition, you can find out from the `AsyncResult` that is returned by the `begin_` method whether the request was sent synchronously or was queued, by calling `sentSynchronously`.

For the generic API, the `sent` method has the following signature:

C#

```
void sent(Ice.AsyncResult r);
```

For the type-safe API, the signature is:

C#

```
void sent(bool sentSynchronously);
```

For the generic API, you can find out whether the request was sent synchronously by calling `sentSynchronously` on the `AsyncResult`. For the type-safe API, the boolean `sentSynchronously` parameter provides the same information.

The `sent` methods allow you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Asynchronous Batch Requests in C#

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws a `System.ArgumentException`.

A batch oneway invocation never calls the generic or type-safe callbacks unless an error occurs before the request is queued. The returned `Ice.AsyncResult` for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The returned result can also be marked completed if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency Semantics for AMI in C#

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or parameter.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch

slice2cs Command-Line Options

The Slice-to-C# compiler, `slice2cs`, offers the following command-line options in addition to the standard options described in Using the Slice Compilers:

- `--tie`
  Generate tie classes.
- `--impl`
  Generate sample implementation files. This option will not overwrite an existing file.
- `--impl-tie`
  Generate sample implementation files using tie classes. This option will not overwrite an existing file.
- `--checksum`
  Generate checksums for Slice definitions.

See Also

- Using the Slice Compilers
- Tie Classes in C-Sharp

Using Slice Checksums in C-Sharp

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2cs`, the `--checksum` option causes the compiler to generate checksums in each C# source file that are added to a member of the `Ice.SliceChecksums` class:

C#

```
namespace Ice
{
    public sealed class SliceChecksums
    {
        public readonly static
System.Collections.Generic.Dictionary<string, string> checksums;
    }
}
```

The `checksums` map is initialized automatically prior to first use; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the dictionaries using the `Equals` function. However, this is not feasible if it is possible that the server might be linked with more Slice definitions than the client. A more general solution is to iterate over the local checksums as demonstrated below:

C#

```
System.Collections.Generic.Dictionary<string,string> serverChecksums =
...
foreach(System.Collections.Generic.KeyValuePair<string, string> e
in Ice.SliceChecksums.checksums)
{
    string checksum;
    if(!serverChecksums.TryGetValue(e.Key, out checksum))
    {
        // No match found for type id!
    }
    else if(!checksum.Equals(e.Value))
    {
        // Checksum mismatch!
    }
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums

Example of a File System Client in C-Sharp

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The C# code hardly differs from the code you would write for an ordinary C# program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local C# object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side C# mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

C#

```
using System;
using Filesystem;

public class Program
```

```
{
    // Recursively print the contents of directory "dir"
    // in tree fashion. For files, show the contents of
    // each file. The "depth" parameter is the current
    // nesting level (for indentation).

    static void listRecursive(DirectoryPrx dir, int depth)
    {
        string indent = new string('\t', ++depth);

        NodePrx[] contents = dir.list();

        foreach (NodePrx node in contents)
            DirectoryPrx subdir = DirectoryPrxHelper.checkedCast(node);
            FilePrx file = FilePrxHelper.uncheckedCast(node);
            Console.WriteLine(
                indent + node.name() +
(subdir != null ? " (directory):" : " (file):"));
            if(subdir != null)
            {
                listRecursive(subdir, depth);
            }
            else
            {
                string[] text = file.read();
                for(int j = 0; j < text.Length; ++j)
                {
                    Console.WriteLine(indent + "\t" + text[j]);
                }
            }
        }
    }

    public static int Main(string[] args)
    {
        try
        {
            using(Ice.Communicator ic = Ice.Util.initialize(ref args))
            {
                //
                // Create a proxy for the root directory
                //
                Ice.ObjectPrx obj
= ic.stringToProxy("RootDir:default -p 10000");

                //
                // Down-cast the proxy to a Directory proxy
                //
                DirectoryPrx rootDir
= DirectoryPrxHelper.checkedCast(obj);
```

```
            //
            // Recursively list the contents of the root directory
            //
            Console.WriteLine("Contents of root directory:");
            listRecursive(rootDir, 0);
        }
    }
    catch(Exception e)
    {
        Console.Error.WriteLine(e);
        return 1;
    }
    return 0;
```

```
        }
    }
```

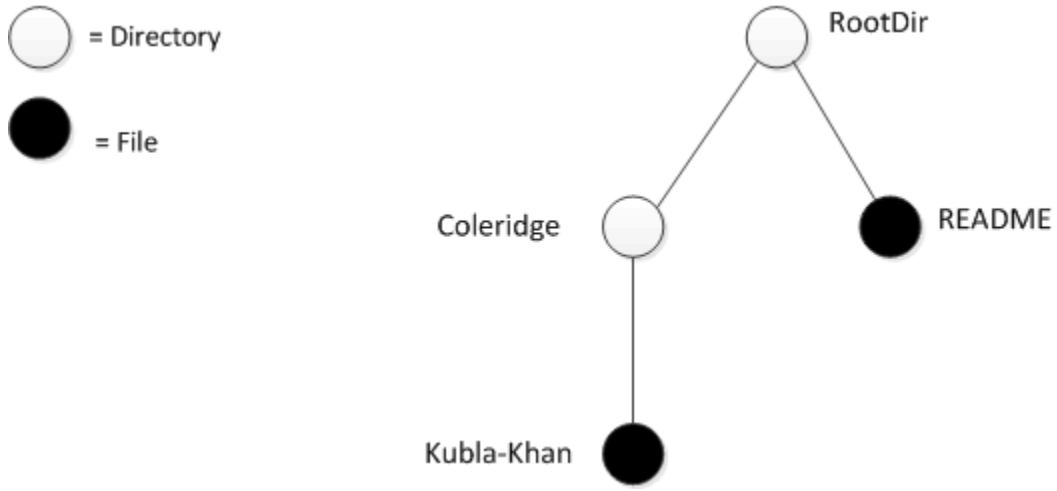
The `Client` class defines two methods: `listRecursive`, which is a helper function to print the contents of the file system, and `Main`, which is the main program. Let us look at `Main` first:

1. The structure of the code in `Main` follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Example of a File System Server in C-Sharp
- Object Life Cycle
- IceGrid

Server-Side Slice-to-C-Sharp Mapping

The mapping for Slice data types to C# is identical on the client side and server side. This means that everything in the Client-Side Slice-to-C-Sharp Mapping also applies to the server side. However, for the server side, there are a few additional things you need to know — specifically, how to:

- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time.

Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Although the examples we present in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any C# mapping rules beyond those described here.

Topics

Server-Side C-Sharp Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing methods in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in C#

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has an abstract method for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
}
```

The Slice compiler generates the following definitions for this interface:

C#

```
namespace Filesystem
{
    public interface NodeOperations_
    {
        string name(Ice.Current current = null);
    }

    public partial interface Node : Ice.Object, NodeOperations_
    {
    }

    public abstract class NodeDisp_ : Ice.ObjectImpl, Node
    {
        public abstract string name(Ice.Current current = null);

        // Mapping-internal code here...
    }
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to C# namespaces with the same name, so the skeleton class definitions are part of the `Filesystem` namespace.
- For each Slice interface *`<interface-name>`*, the compiler generates the C# interface *`<interface-name>`*`Operations_` (`NodeOperations_` in this example). This interface contains a method for each operation in the Slice interface. (You can ignore the `Ice.Current` parameter for now.)
- For each Slice interface *`<interface-name>`*, the compiler generates a C# interface *`<interface-name>`* (`Node` in this example). That interface extends `Ice.Object` and the operations interface.
- For each Slice interface *`<interface-name>`*, the compiler generates an abstract class *`<interface-name>`*`Disp_` (`NodeDisp_` in this example). This abstract class is the actual skeleton class; it is the base class from which you derive your servant class.

> The `Operations_` interface is used by tie classes. Without them, there would be no need for this additional interface.

`Ice.Object` Base Interface for C# Servants

`Object` is mapped to the `Ice.Object` interface in C#:

C#

```
namespace Ice
{
    public interface Object : ICloneable
    {
        bool ice_isA(string s, Current current = null);
        void ice_ping(Current current = null);
        string[] ice_ids(Current current = null);
        string ice_id(Current current = null);

        Task<OutputStream> ice_dispatch(Request request);

        ...
    }
}
```

The methods of `Ice.Object` behave as follows:

- `ice_isA`
  This method returns `true` if target object implements the given type ID, and `false` otherwise.
- `ice_ping`
  `ice_ping` provides a basic reachability test for the servant.
- `ice_ids`
  This method returns a string array representing all of the type IDs implemented by this servant, including `::Ice::Object`.
- `ice_id`
  This method returns the type ID of the most-derived interface implemented by this servant.
- `ice_dispatch`
  This method dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.

Servant Classes in C#

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

C#

```
public class NodeI : NodeDisp_
{
    public NodeI(string name)
    {
        _name = name;
    }

    public override string name(Ice.Current current)
    {
        return _name;
    }

    private string _name;
}
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.) Note that `NodeI` extends `NodeDisp_`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the abstract `name` method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other methods and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` method returns its value.)

Server-Side Normal and `idempotent` Operations in C#

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

Slice

```
interface Example
{
   void              normalOp();
   idempotent void   idempotentOp();
}
```

The operations class for this interface looks like this:

C#

```
public interface ExampleOperations_
{
    void normalOp(Ice.Current current = null);
    void idempotentOp(Ice.Current current = null);
}
```

Note that the signatures of the methods are unaffected by the `idempotent` qualifier.

See Also

- Slice for a Simple File System
- Parameter Passing in C-Sharp
- Raising Exceptions in C-Sharp
- Tie Classes in C-Sharp
- The Current Object

Parameter Passing in C-Sharp

Parameter passing on the server side follows the rules for the client side. Additionally, every operation receives a trailing parameter of type `Current`. For example, the `name` operation of the `Node` interface has no parameters, but the corresponding `name` method of the servant interface has a single parameter of type `Current`. We will ignore this parameter for now.

> The parameter-passing rules change somewhat when using the asynchronous mapping.

On this page:

Server-Side Mapping for Parameters in C#

The servant mapping for operations is consistent with the proxy mapping. To illustrate the rules for the C# mapping, consider the following interface:

Slice

```
module M
{
    interface Example
    {
        string op(string sin, out string sout);
    }
}
```

The generated method for `op` looks as follows:

C#

```
public interface ExampleOperations_
{
    string op(string sin, out string sout, Ice.Current current = null);
}
```

As you can see, there are no surprises here. For example, we could implement `op` as follows:

C#

```
using System;

public class ExampleI : ExampleDisp_
{
    public override string op(string sin, out string sout,
Ice.Current current = null)
    {
        Console.WriteLine(sin);      // In params are initialized
        sout = "Hello World!";       // Assign out param
        return "Done";
    }
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a method; the fact that remote procedure calls are involved does not affect your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal C# rules and do not require special-purpose API calls.

Thread-Safe Marshaling in C#

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For C# applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following servant implementation:

C#

```
public class GridI : GridDisp_
{
    GridI()
    {
        _grid = // ...
    }

    public override int[][] getGrid(Current current = null)
    {
        return _grid;
    }

    public override void setValue(int x, int y, int val, Current current
= null)
    {
        _grid[x][y] = val;
    }

    private int[][] _grid;
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned array in preparation to send a reply message, it is possible for another thread to dispatch the `setValue` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

C#

```
public class GridI : GridDisp_
{
    public override int[][] getGrid(Current current = null)
    {
        lock(this)
        {
            return _grid;
        }
    }

    public override void setValue(int x, int y, int val, Current current
= null)
    {
        lock(this)
        {
            int[][] newGrid = // shallow copy...
            newGrid[x][y] = val;
            _grid = newGrid;
        }
    }

    ...
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `marshaled-result` metadata directive changes the signature of the corresponding servant method, but only if that operation returns mutable types. The metadata directive has the following effects:

- For an operation `op` from an interface `Intf` that returns one or multiple values and at least one of those values has a mutable

type, the Slice compiler generates an `Intf_OpMarshaledResult` class and the return type of the servant method becomes `OpMarshaledResult`.

- The constructor for `Intf_OpMarshaledResult` takes an extra argument of type `Current`. The servant must supply the `Current` in order for the results to be marshaled correctly.

The metadata directive also affects the asynchronous mapping but has no effect on the proxy mapping, nor does it affect the servant mapping of Slice operations that return `void` or return only immutable values.

> You can also annotate an interface with the `marshaled-result` metadata and it will be applied to all of the interface's operations.

After applying the metadata, we can now implement the `Grid` servant as follows:

C#

```
public class GridI : GridDisp_
{
   public override Grid_GetGridMarshaledResult getGrid(Current current)
   {
        lock(this)
        {
            return new Grid_GetGridMarshaledResult(_grid, current); //
_grid is marshaled immediately
        }
    }

    public override void setValue(int x, int y, int val, Current
current)
    {
        lock(this)
        {
            _grid[x][y] = val; // this is safe
        }
    }

    ...
}
```

Here are more examples to demonstrate the mapping:

Slice

```
class C { ... }
struct S { ... }
sequence<string> Seq;

interface Example
{
    C getC();
    ["marshaled-result"]
    C getC2();

    void getS(out S val);

    ["marshaled-result"]
    void getS2(out S val);

    string getValues(string name, out Seq val);

    ["marshaled-result"]
    string getValues2(string name, out Seq val);

    ["amd", "marshaled-result"]
    string getValuesAMD(string name, out Seq val);
}
```

Review the generated code below to see the changes that the presence of the metadata causes in the servant method signatures:

C#

```
public struct Example_GetC2MarshaledResult : Ice.MarshaledResult
{
    public Example_GetC2MarshaledResult(C returnValue, Current current);
    ...
}

public struct Example_GetS2MarshaledResult : Ice.MarshaledResult
{
    public Example_GetS2MarshaledResult(S returnValue, Current current);
    ...
}

public struct Example_GetValues2MarshaledResult : Ice.MarshaledResult
{
    public Example_GetValues2MarshaledResult(string returnValue,
string[] val, Current current);
    ...
}

public struct Example_GetValuesAMDResult
{
    public Example_GetValuesAMDResult(string returnValue, string[] val);
    ...
}

public struct Example_GetValuesAMDMarshaledResult : Ice.MarshaledResult
{
    public Example_GetValuesAMDMarshaledResult(string ret, string[] val,
Ice.Current current);
    ...
}

public interface ExampleOperations_
{
    C getC(Ice.Current current = null);
    Example_GetC2MarshaledResult getC2(Ice.Current current = null);
    S getS(Ice.Current current = null);
    Example_GetS2MarshaledResult getS2(Ice.Current current = null);
    string getValues(string name, out string[] val, Ice.Current current
= null);
    Example_GetValues2MarshaledResult getValues2(string name,
Ice.Current current = null);
    System.Threading.Tasks.Task<M.Example_GetValuesAMDMarshaledResult>
getValuesAMDAsync(string name, Ice.Current current = null);
}
```

See Also

- Server-Side C-Sharp Mapping for Interfaces
- Raising Exceptions in C-Sharp
- Tie Classes in C-Sharp
- The Current Object

Raising Exceptions in C-Sharp

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

C#

```
// ...
public override void write(string[] text, Ice.Current current)
{
    try
    {
        // Try to write file contents here...
    }
    catch(System.Exception ex)
    {
        GenericError e = new GenericError("cannot write file", ex);
        e.reason = "Exception during write operation";
        throw e;
    }
}
```

Note that, for this example, we have supplied the optional second parameter to the `GenericError` constructor. This parameter sets the `InnerException` member of `System.Exception` and preserves the original cause of the error for later diagnosis.

If you throw an arbitrary C# run-time exception (such as an `InvalidCastException`), the Ice run time catches the exception and then returns an `UnknownException` to the client.

The server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw an Ice run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

See Also

- Run-Time Exceptions
- C-Sharp Mapping for Exceptions
- Server-Side C-Sharp Mapping for Interfaces
- Parameter Passing in C-Sharp
- Tie Classes in C-Sharp

Tie Classes in C-Sharp

The mapping to skeleton classes requires the servant class to inherit from its skeleton class. Occasionally, this creates a problem: some class libraries require you to inherit from a base class in order to access functionality provided by the library; because C# does not support multiple implementation inheritance, this means that you cannot use such a class library to implement your servants because your servants cannot inherit from both the library class and the skeleton class simultaneously.

To allow you to still use such class libraries, Ice provides a way to write servants that replaces inheritance with delegation. This approach is supported by *tie classes*. The idea is that, instead of inheriting from the skeleton class, you simply create a class (known as an *implementation class* or *delegate class*) that contains methods corresponding to the operations of an interface. You use `["cs:tie"]` metadata directive to create a tie class. For example, adding this directive in front of the `Node` interface we saw previously makes `slice2cs` emit an additional tie class. For an interface `<interface-name>`, the generated tie class has the name `<interface-name>Tie_`:

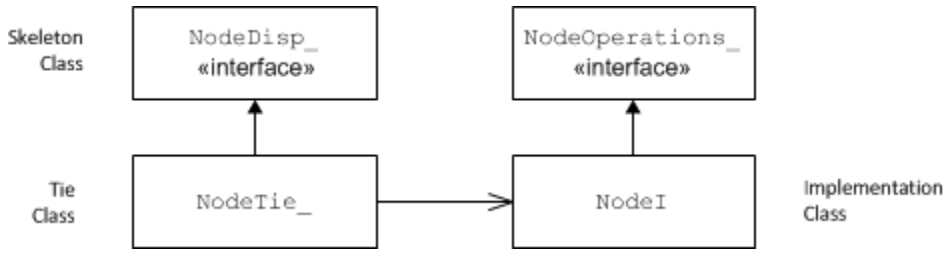
Slice

```
module Filesystem
{
    ["cs:tie"] interface Node
    {
        ...
    }
}
```

C#

```
public class NodeTie_ : NodeDisp_, Ice.TieBase
{
    public NodeTie_()
    {
    }

    public NodeTie_(NodeOperations_ del)
    {
        _ice_delegate = del;
    }

    public object ice_delegate()
    {
        return _ice_delegate;
    }

    public void ice_delegate(object del)
    {
        _ice_delegate = (NodeOperations_)del;
    }

    public override int GetHashCode()
    {
        return _ice_delegate == null ? 0 : _ice_delegate.GetHashCode();
    }

    public override bool Equals(object rhs)
    {
       if(object.ReferenceEquals(this, rhs))
       {
           return true;
       }
       if(!(rhs is NodeTie_))
       {
           return false;
       }
       if(_ice_delegate == null)
       {
           return ((NodeTie_)rhs)._ice_delegate == null;
       }
       return _ice_delegate.Equals(((NodeTie_)rhs)._ice_delegate);
    }

    public override string name(Ice.Current current)
    {
        return _ice_delegate.name(current);
    }

    private NodeOperations_ _ice_delegate;
}
```

This looks a lot worse than it is: in essence, the generated tie class is simply a servant class (it extends `NodeDisp_`) that delegates each invocation of a method that corresponds to a Slice operation to your implementation class:



A skeleton class, tie class, and implementation class.

The `Ice.TieBase` interface defines the `ice_delegate` methods that allow you to get and set the delegate.

Given this machinery, we can create an implementation class for our `Node` interface as follows:

C#

```
public class NodeI : NodeOperations_
{
    public NodeI(string name)
    {
        _name = name;
    }

    public override string name(Ice.Current current)
    {
        return _name;
    }

    private string _name;
}
```

Note that this class is identical to our previous implementation, except that it implements the `NodeOperations_` interface and does not extend `NodeDisp_` (which means that you are now free to extend any other class to support your implementation).

To create a servant, you instantiate your implementation class and the tie class, passing a reference to the implementation instance to the tie constructor:

C#

```
NodeI fred = new NodeI("Fred");         // Create implementation
NodeTie_ servant = new NodeTie_(fred);  // Create tie
```

Alternatively, you can also default-construct the tie class and later set its delegate instance by calling `ice_delegate`:

C#

```
NodeTie_ servant = new NodeTie_();      // Create tie
// ...
NodeI fred = new NodeI("Fred");         // Create implementation
// ...
servant.ice_delegate(fred);             // Set delegate
```

When using tie classes, it is important to remember that the tie instance is the servant, not your delegate. Furthermore, you must not use a tie instance to incarnate an Ice object until the tie has a delegate. Once you have set the delegate, you must not change it for the lifetime of the tie; otherwise, undefined behavior results.

You should use the tie approach only if you need to, that is, if you need to extend some base class in order to implement your servants: using the tie approach is more costly in terms of memory because each Ice object is incarnated by two C# objects (the tie and the delegate) instead of just one. In addition, call dispatch for ties is marginally slower than for ordinary servants because the tie forwards each operation to the delegate, that is, each operation invocation requires two function calls instead of one.

Also note that, unless you arrange for it, there is no way to get from the delegate back to the tie. If you need to navigate back to the tie from the delegate, you can store a reference to the tie in a member of the delegate. (The reference can, for example, be initialized by the constructor of the delegate.)

See Also

- Server-Side C-Sharp Mapping for Interfaces
- Parameter Passing in C-Sharp
- Raising Exceptions in C-Sharp
- Object Incarnation in C-Sharp

Object Incarnation in C-Sharp

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a C# Servant

Instantiating a servant means to allocate an instance:

C#

```
Node servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance and assigns its address to a reference of type `Node`. This works because `NodeI` is derived from `Node`, so a `Node` reference can refer to an instance of type `NodeI`. However, if we want to invoke a method of the `NodeI` class at this point, we must use a `NodeI` reference:

C#

```
NodeI servant = new NodeI("Fred");
```

Whether you use a `Node` or a `NodeI` reference depends purely on whether you want to invoke a method of the `NodeI` class: if not, a `Node` reference works just as well as a `NodeI` reference.

Creating an Identity in C#

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

C#

```
Ice.Identity id = new Ice.Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

Activating a C# Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

C#

```
_adapter.add(servant, id);
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct method on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

UUIDs as Identities in C#

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. .NET provides a helper function that we can use to create such identities:

C#

```
public class Example
{
    public static void Main(string[] args)
    {
        System.Console.WriteLine(System.Guid.NewGuid().ToString());
    }
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `NewGuid` creates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

C#

```
_adapter.addWithUUID(new NodeI("Fred"));
```

Creating Proxies in C#

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Hello World Application. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in C#

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

C#

```
NodePrx proxy = NodePrxHelper.uncheckedCast(_adapter.addWithUUID(new No
deI("Fred")));
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

Direct Proxy Creation in C#

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

C#

```
Ice.Identity id = new Ice.Identity();
id.name = System.Guid.NewGuid().ToString();
Ice.ObjectPrx o = _adapter.createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `NewGuid`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Hello World Application
- Server-Side C-Sharp Mapping for Interfaces
- Object Adapter States
- Servant Locators
- Object Life Cycle

Asynchronous Method Dispatch (AMD) in C-Sharp

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool. If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server can provide its results to the Ice run time for delivery to the client.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

On this page:

Enabling AMD with Metadata in C#

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive replaces synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

Slice

```
["amd"] interface I
{
    bool isValid();
    float computeRate();
}

interface J
{
    ["amd"] void startProcess();
    int endProcess();
}
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface level) minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

AMD Mapping in C#

The asynchronous mapping for an operation differs in several ways from its synchronous mapping:

- The dispatch method name has the suffix `Async`
- For an operation that returns `void` and has no out parameters, the return type of the dispatch method is `System.Threading.Tasks.Task`
- For an operation that returns at least one value, the dispatch method returns `System.Threading.Tasks.Task<T>`, where `T` represents the return type as described below
- The dispatch method does not declare any out parameters

Let's start with some simple examples to demonstrate the asynchronous mapping:

Slice

```
["amd"]
interface Example
{
    void opVoid(int n);
    string opString();
    void opStringOut(out string s);
}
```

The Slice-to-C# compiler generates the following base class:

C#

```
public abstract class ExampleDisp_ : Ice.ObjectImpl, Example
{
    public abstract System.Threading.Tasks.Task opVoidAsync(int n,
Ice.Current current = null);

    public abstract System.Threading.Tasks.Task<string>
opStringAsync(Ice.Current current = null);

    public abstract System.Threading.Tasks.Task<string>
opStringOutAsync(Ice.Current current = null);
    ...
}
```

Pay particular attention to the mappings for `opString` and `opStringOut`. For operations like these that return a single value of type `T` (whether it's a non-`void` return value or an out parameter), the method returns `Task<T>`.

Finally, for an operation that returns multiple values, the Slice compiler generates an additional structure to hold the results. This "result type" is shared between the asynchronous proxy mapping and the asynchronous dispatch mapping. Refer to the AMI discussion for details of the mapping for result types.

Let's add another operation to our example to demonstrate the mapping for multiple return values:

Slice

```
["amd"]
interface Example
{
    void opVoid(int n);
    string opString();
    void opStringOut(out string s);
    string opAll(bool flag, out int count);
}
```

The mapping for `opAll` is shown below:

C#

```
public struct Example_OpAllResult
{
    public Example_OpAllResult(string returnValue, int count)
    {
        this.returnValue = returnValue;
        this.count = count;
    }
    public string returnValue;
    public int count;
}

public abstract class ExampleDisp_ : Ice.ObjectImpl, Example
{
    ...
    public abstract System.Threading.Tasks.Task<Example_OpAllResult>
opAllAsync(bool flag, Ice.Current current = null);
    ...
}
```

As you can see, the Slice compiler generated the `Example_OpAllResult` structure to encapsulate the results of an asynchronous invocation of `opAll`.

AMD Tasks in C#

A servant's implementation of an asynchronous dispatch method is responsible for providing a `Task` object that must eventually complete successfully or raise an exception.

> Failing to complete a task will cause the client's invocation to hang because no response will ever be sent.

Using .NET's task-based asynchronous pattern provides servants with a lot of flexibility in their implementations. Here's a very simple example:

C#

```
using System.Threading.Tasks;

public class ExampleI : ExampleDisp_
{
    public override Task<string> opStringAsync(Ice.Current current)
    {
        return Task.FromResult<string>("hello world!");
    }
}
```

The `Task.FromResult` method produces an already-completed task with the given result value. Clearly this implementation isn't taking advantage of asynchronous programming at all; it's just a more complex version of a synchronous implementation. Here's a slightly more interesting version:

C#

```
using System.Threading.Tasks;

public class ExampleI : ExampleDisp_
{
    public override Task<string> opStringAsync(Ice.Current current)
    {
        return Task<string>.Run(() =>
            {
                longRunningOperation();
                return "hello world!";
            });
    }
}
```

We've called `Task.Run` to create a new task and supplied a lambda as our task implementation. The dispatch method returns the task to the Ice run time. Upon completion of the task, Ice will return the response to the client. We can simplify this example using the `async` and `await` keywords as follows:

C#

```
using System.Threading.Tasks;

public class ExampleI : ExampleDisp_
{
    public override async Task<string> opStringAsync(Ice.Current
current)
    {
        await longRunningOperation();
        return "hello world!";
    }
}
```

These examples are just scratching the surface of what's possible for asynchronous dispatch implementations.

AMD Thread Safety in C#

As with the synchronous mapping, you can add the `marshaled-result` metadata to operations that return mutable types in order to avoid potential thread-safety issues. The return type of your operation will change to be `Task<`*Op*`MarshaledResult>`.

Chaining AMI and AMD Invocations in C#

Since the asynchronous proxy API and the asynchronous dispatch API both use tasks, chaining nested invocations together without blocking Ice's thread pool threads becomes very straightforward. Continuing our example from the previous section, suppose our servant also holds a proxy to another object of the same type and derives its response from that of the other object:

C#

```
using System.Threading.Tasks;

public class ExampleI : ExampleDisp_
{
    public override Task<string> opStringAsync(Ice.Current current)
    {
        return other.opStringAsync().ContinueWith((task) =>
            {
                return "hello " + task.Result;
            });
    }

    private ExamplePrx other;
}
```

We've initiated an asynchronous proxy invocation on the other object and used a lambda continuation to form our result. Here's a much simpler version that uses the `async` and `await` keywords:

C#

```
using System.Threading.Tasks;

public class ExampleI : ExampleDisp_
{
    public override async Task<string> opStringAsync(Ice.Current
current)
    {
        return "hello " + await other.opStringAsync();
    }

    private ExamplePrx other;
}
```

The Ice dispatch thread is *not* blocked by the nested call to `opStringAsync`. Rather, the use of `await` causes the dispatch thread to be released back to Ice, and the remainder of the code will be executed as a continuation when Ice receives the reply.

AMD Exceptions in C#

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation and calls the servant's dispatch method), and the response thread (the thread that completes the task).

> These are not necessarily two different threads: it is legal for the dispatch method to return an already-completed task.

Although we recommend that the task be used to return all exceptions to the client, it is legal for the dispatch method to raise an exception directly.

AMD Example in C#

For a more realistic example of using AMD in Ice, let's define the Slice interface for a simple computational engine:

Slice

```
module Demo
{
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {}

    interface Model
    {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    }
}
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo.ModelDisp_` and supplies a definition for the `interpolateAsync` method that adds a job to a work queue. The queue uses a `Job` object to hold the arguments and perform the interpolation (not shown). The job also creates a task in which the interpolation is performed.

C#

```
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Generic;

public class ModelI : Demo.ModelDisp_
{
    public override Task<float[][]> interpolateAsync(float[][] data,
float factor, Ice.Current current)
    {
        return _workQueue.add(data, factor);
    }

    private WorkQueue _workQueue = ...;
}

class WorkQueue {
    public Task<float[][]> add(float[][] data, float factor)
    {
        lock(this)
        {
            Job j = new Job(data, factor);
            _jobs.AddLast(j);
            Monitor.Pulse(this);
            return j.task();
        }
    }

    private void Dispatch() // Runs in a separate thread
    {
        while(true)
        {
            lock(this)
            {
                Monitor.Wait(this);
                Job job = _jobs.First.Value;
                _jobs.RemoveFirst();
                job.task().Start();
                jot.task().Wait();
            }
        }
    }

    private LinkedList<Job> _jobs = new LinkedList<Job>();
```

```
class Job
{
    public Job(float[][] data, float factor)
    {
        _data = data;
        _factor = factor;
        _task = new Task<float[][]>(() => return execute());
    }

    public Task<float[][]> task()
    {
        return _task;
    }

    private float[][] execute()
    {
        if(!interpolateGrid())
        {
            throw new Demo.RangeError();
        }
        else
        {
            return _data;
        }
    }

    private boolean interpolateGrid()
    {
        // ...
    }

    private float[][] _grid;
    private float _factor;
    private Task<float[][]> _task;
```

```
        }
    }
```

If `interpolateGrid` returns `false`, the task raises an exception to indicate that a range error has occurred. If interpolation was successful, the task returns the modified grid as its result.

See Also

- User Exceptions
- Asynchronous Method Invocation (AMI) in C-Sharp
- The Ice Threading Model

Example of a File System Server in C-Sharp

This page presents the source code for a C# server that implements our file system and communicates with the client we wrote earlier. The code is fully functional, apart from the required interlocking for threads.

The server is free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

On this page:

Implementing a File System Server in C#

We have now seen enough of the server-side C# mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of three source files:

- `Program.cs`
  This file contains the server main program.
- `DirectoryI.cs`
  This file contains the implementation for the `Directory` servants.
- `FileI.cs`
  This file contains the implementation for the `File` servants.

Server Main Program in C#

Our server main program, in the file `Program.cs`, consists of two static methods, `Main` and `run`. `Main` creates and destroys an Ice communicator, and `run` uses this communicator instantiate our file system objects:

C#

```
using Filesystem;
using System;

public class Program
{
    public static int Main(string[] args)
    {
        int status = 0;

        try
        {
            //
            // using statement - communicator is automatically destroyed
            // at the end of this statement
            //
            using(var communicator = Ice.Util.initialize(ref args))
            {
```

```
                //
                // Destroy the communicator on Ctrl+C or Ctrl+Break
                //
                Console.CancelKeyPress += (sender, eventArgs) =>
communicator.destroy();

                status = run(communicator);
            }
        }
        catch(Exception ex)
        {
            Console.Error.WriteLine(ex);
            status = 1;
        }

        return status;
    }

    private static int run(Ice.Communicator communicator)
    {
        //
        // Create an object adapter.
        //
        var adapter =

communicator.createObjectAdapterWithEndpoints("SimpleFilesystem",
"default -h localhost -p 10000");

        //
        // Create the root directory (with name "/" and no parent)
        //
        var root = new DirectoryI(communicator, "/", null);
        root.activate(adapter);

        //
        // Create a file called "README" in the root directory
        //
        var file = new FileI(communicator, "README", root);
        var text = new string[]{ "This file system contains a collection
of poetry." };
        try
        {
            file.write(text);
        }
        catch(GenericError e)
        {
            Console.Error.WriteLine(e.reason);
        }
        file.activate(adapter);
```

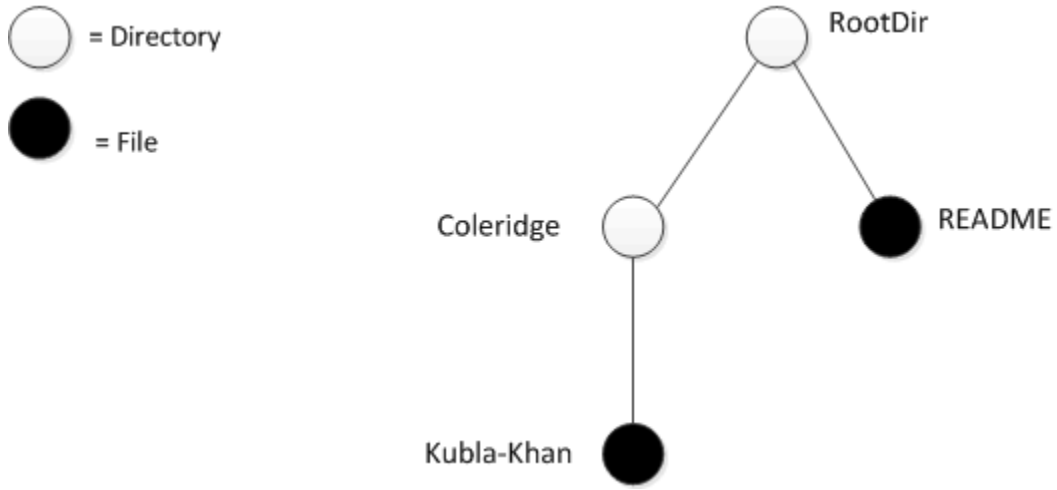
```
//
// Create a directory called "Coleridge" in the root directory
//
var coleridge = new DirectoryI(communicator, "Coleridge", root);
coleridge.activate(adapter);

//
// Create a file called "Kubla_Khan" in the Coleridge directory
//
file = new FileI(communicator, "Kubla_Khan", coleridge);
text = new string[]{ "In Xanadu did Kubla Khan",
                     "A stately pleasure-dome decree:",
                     "Where Alph, the sacred river, ran",
                     "Through caverns measureless to man",
                     "Down to a sunless sea." };
try
{
    file.write(text);
}
catch(GenericError e)
{
    Console.Error.WriteLine(e.reason);
}
file.activate(adapter);

//
// All objects are created, allow client requests now
//
adapter.activate();

//
// Wait until we are done
//
communicator.waitForShutdown();

return 0;
```

```
        }
    }
```

The code uses a `using` directive for the `Filesystem` namespace. This avoids having to continuously use fully-qualified identifiers with a `Filesystem.` prefix.

The next part of the source code is the definition of the `Program` class. Much of this code is boiler plate: we create a communicator, then an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

C#

```
DirectoryI root = new DirectoryI("/", null);
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in the above illustration:

C#

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI("/", null);

// Create a file called "README" in the root directory
//
File file = new FileI("README", root);
string[] text;
text = new string[]
{
    "This file system contains a collection of poetry."
};
try
{
    file.write(text);
}
catch(GenericError e)
{
    Console.Error.WriteLine(e.reason);
}

// Create a directory called "Coleridge"
// in the root directory
//
DirectoryI coleridge = new DirectoryI("Coleridge", root);

// Create a file called "Kubla_Khan"
// in the Coleridge directory
//
file = new FileI("Kubla_Khan", coleridge);
text = new string[]
{
  "In Xanadu did Kubla Khan",
  "A stately pleasure-dome decree:",
  "Where Alph, the sacred river, ran",
  "Through caverns measureless to man",
  "Down to a sunless sea."
};
try
{
    file.write(text);
}
catch (GenericError e)
{
    Console.Error.WriteLine(e.reason);
}
```

We first create the root directory and a file `README` within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

C#

```
string[] text;
text = new string[]
{
    "This file system contains a collection of poetry."
}
try
{
    file.write(text);
}
catch (GenericError e)
{
    Console.Error.WriteLine(e.reason);
}
```

Recall that Slice sequences by default map to C# arrays. The Slice type `Lines` is simply an array of strings; we add a line of text to our `README` file by initializing the `text` array to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by writing:

C#

```
file.write(text);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and not via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary C# method call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the above illustration.

`FileI` Servant Class in C#

Our `FileI` servant class has the following basic structure:

C#

```
using Filesystem;
using System;

public class FileI : FileDisp_
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private string _name;
    private DirectoryI _parent;
    private string[] _lines;
}
```

The class has a number of data members:

- `_adapter`
  This static member stores a reference to the single object adapter we use in our server.
- `_name`
  This member stores the name of the file incarnated by the servant.
- `_parent`
  This member stores the reference to the servant for the file's parent directory.
- `_lines`
  This member holds the contents of the file.

The `_name` and `_parent` data members are initialized by the constructor:

C#

```
    public FileI(string name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;

        Debug.Assert(_parent != null);

        // Create an identity
        //
        Ice.Identity myID = new Ice.Identity();
        myId.name = System.Guid.NewGuid().ToString();

        // Add the identity to the object adapter
        //
        _adapter.add(this, myID);

        // Create a proxy for the new node and
        // add it as a child to the parent
        //
        NodePrx thisNode = NodePrxHelper.uncheckedCast(_adapter.createP
roxy(myID));
        _parent.addChild(thisNode);
    }
```

After initializing the `_name` and `_parent` members, the code verifies that the reference to the parent is not null because every file must have a parent directory. The constructor then generates an identity for the file by calling `NewGuid` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function in `DirectoryI` Methods.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

C#

```
    // Slice Node::name() operation

    public override string name(Ice.Current current)
    {
        return _name;
    }

    // Slice File::read() operation

    public override string[] read(Ice.Current current)
    {
        return _lines;
    }

    // Slice File::write() operation

    public override void write(string[] text, Ice.Current current)
    {
        _lines = text;
    }
```

The `name` method is inherited from the generated `Node` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived). It simply returns the value of the `_name` member.

The `read` and `write` methods are inherited from the generated `File` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived) and simply return and set the `_lines` member.

`DirectoryI` Servant Class in C#

The `DirectoryI` class has the following basic structure:

C#

```
using Filesystem;
using System;
using System.Collections;

public class DirectoryI : DirectoryDisp_
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private string _name;
    private DirectoryI _parent;
    private ArrayList _contents = new ArrayList();
}
```

DirectoryI Data Members

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds a null reference.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

C#

```
    public DirectoryI(string name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;

        // Create an identity. The
        // parent has the fixed identity "RootDir"
        //
        Ice.Identity myID = new Ice.Identity();
        myID.name = _parent != null ? System.Guid.NewGuid().ToString()
: "RootDir";

        // Add the identity to the object adapter
        //
        _adapter.add(this, myID);

        // Create a proxy for the new node and
        // add it as a child to the parent
        //
        NodePrx thisNode = NodePrxHelper.uncheckedCast(_adapter.createP
roxy(myID));
        if(_parent != null)
        {
            _parent.addChild(thisNode);
        }
    }
```

DirectoryI Constructor

The constructor creates an identity for the new directory by calling `NewGuid`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a proxy to itself and passes it to the `addChild` helper function.

DirectoryI Methods

`addChild` simply adds the passed reference to the `_contents` list:

C#

```
public void addChild(NodePrx child)
{
    _contents.Add(child);
}
```

The remainder of the operations, `name` and `list`, are trivial:

C#

```
public override string name(Ice.Current current)
{
    return _name;
}

public override NodePrx[] list(Ice.Current current)
{
    return (NodePrx[])_contents.ToArray(typeof(NodePrx));
}
```

Note that the `_contents` member is of type `System.Collections.ArrayList`, which is convenient for the implementation of the `addChild` method. However, this requires us to convert the list into a C# array in order to return it from the `list` operation.

See Also

- Slice for a Simple File System
- Example of a File System Client in C-Sharp
- C-Sharp Mapping for Sequences
- The Ice Threading Model

Slice-to-C-Sharp Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to C# is identical to the mapping for these constructs without the `local` qualifier. The generated C# code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

The rest of this section describes the mapping of the remaining local types to C#:

- C-Sharp Mapping for Local Interfaces
- C-Sharp Mapping for Local Classes
- C-Sharp Mapping for Local Exceptions
- C-Sharp Mapping for Operations on Local Types
- C-Sharp Mapping for Data Members in Local Types

C-Sharp Mapping for Local Interfaces

On this page:

Mapped C# Class

A Slice local interface is mapped to a C# interface with the same name, for example:

Slice

```
module Ice
{
    local interface Communicator
    {
        ...
    }
}
```

is mapped to the C# interface `Communicator`:

C#

```
namespace Ice
{
    public partial interface Communicator
    {
        ...
    }
}
```

The `delegate` metadata allows you to map a local interface with a single operation to a C# delegate. For example:

Slice

```
module Ice
{
    ["delegate"] local interface ValueFactory
    {
        Value create(string type);
    }
}
```

is mapped to:

C#

```
namespace Ice
{
    public delegate Value ValueFactory(string type);
}
```

LocalObject in C#

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to System`.Object` in C#.

Mapping for Local Interface Inheritance in C#

Inheritance of local Slice interfaces is mapped to interface inheritance in C#. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

C#

```
namespace M
{
    public partial interface A {}
    public partial interface B : A {}
    public interface C : A {}
    public interface D : B, C {}
}
```

C-Sharp Mapping for Local Classes

On this page:

Mapped C# Class

A local Slice class is mapped to a concrete or abstract C# class with the same name. For example:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ...
    }
}
```

is mapped to the C# class `ConnectionInfo`:

C#

```
namespace Ice
{
    public partial class ConnectionInfo
    {
        ...
    }
}
```

LocalObject in C#

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to `System.Object` in C#.

Mapping for Local Interface Inheritance in C#

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. Both `extends` and `implements` are mapped to inheritance in C#. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

C#

```
namespace M
{
    public partial interface A {}
    public partial interface B {}

    public partial class C : A, B {}
    public partial class D : C {}
}
```

C-Sharp Mapping for Local Exceptions

On this page:

Mapped C# Class

A local Slice exception is mapped to a C# class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the C# class `InitializationException`:

C#

```
namespace Ice
{
    public partial class InitializationException : LocalException
    {
        ...
    }
}
```

Base Class for Local Exceptions in Java

All mapped C# classes for local exceptions derive from `LocalException`:

C#

```
namespace Ice
{
    [Serializable]
    public abstract class LocalException : Exception
    {
        public LocalException() {}
        public LocalException(System.Exception ex) { ... }
        ...
    }
}
```

Mapping for Local Exception Inheritance in C#

A local Slice exception can extend another Slice exception, which is mapped to class inheritance in C#. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

C#

```
namespace M
{
    public partial class ErrorBase : Ice.LocalException { ... }
    public partial class ResourceError : ErrorBase { ... }
}
```

C-Sharp Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to a C# method with the same name. The mapping of operation parameters to C# is identical to the Client-Side Mapping for these parameters.

Unlike the Client-Side mapping, the mapped method does not have a trailing `Ice.OptionalContext` parameter.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

C#

```
namespace M
{
    public partial interface L
    {
        string op(int n, string s, _System.Object any, out int m, out
string t, out M.L[] newLSeq);
    }
}
```

C-Sharp Mapping for Data Members in Local Types

Data members on local Slice types (classes, exceptions and structs) are mapped to C# just like the data members of the corresponding non local Slice construct.

A local Slice type can have a data member of type local interface or class: it is mapped a C# data member with the mapped C# interface or class as its type.

The .NET Utility Library

Ice for C# includes a number of utility APIs in the `Ice.Util` class. This appendix summarizes the contents of these APIs for your reference.

On this page:

Communicator Initialization Methods

`Ice.Util` provides a number of overloaded `initialize` methods that create a communicator.

Identity Conversion

`Ice.Util` contains two methods to convert object identities of type `Ice.Identity` to and from strings.

Per-Process Logger Methods

`Ice.Util` provides methods for getting and setting the per-process logger.

Property Creation Methods

`Ice.Util` provides a number of overloaded `createProperties` methods that create property sets.

Proxy Comparison Methods

Two methods, `proxyIdentityCompare` and `proxyIdentityAndFacetCompare`, allow you to compare object identities that are stored in proxies (either ignoring the facet or taking the facet into account).

Version Information

The `stringVersion` and `intVersion` methods return the version of the Ice run time:

C#

```
public static string stringVersion();
public static int intVersion();
```

The `stringVersion` method returns the Ice version in the form <*major*>.<*minor*>.<*patch*>, for example, `3.7.1`. For beta releases, the version is <*major*>.<*minor*>b, for example, `3.7b`.

The `intVersion` method returns the Ice version in the form *AABBCC*, where *AA* is the major version number, *BB* is the minor version number, and *CC* is patch level, for example, `30701` for version 3.7.1. For beta releases, the patch level is set to 51 so, for example, for version 3.7b, the value is `30751`.

See Also

- Command-Line Parsing and Initialization
- Setting Properties
- C-Sharp Streaming Interfaces

- Object Identity

Java Mapping

Topics

Selecting the Java Mapping

Ice provides two distinct Java mappings:

- Java Compat
  This mapping is largely backward-compatible with prior Ice releases. Although Ice 3.7 no longer supports Java versions prior to Java 8, the "Compat" mapping does not depend on any Java 8-specific language or run-time features.

- Java
  This is a new mapping that takes advantage of features in Java 8.

This chapter describes the Java mapping.

Selecting the Java Mapping

slice2java, the Slice-to-Java translator, generates code for the Java mapping by default.

Initialization in Java

Every Ice-based application needs to initialize the Ice run time, and this initialization returns a `com.zeroc.Ice.Communicator` object.

A `Communicator` is a local Java object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application.

You initialize the Ice run time by calling `com.zeroc.Ice.Util.initialize`, for example:

Java

```
public static void main(String[] args)
{
    com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(args);
    ...
}
```

`Util.initialize` accepts the argument vector that is passed to `main` by the operating system. The method scans the argument vector for any command-line options that are relevant to the Ice run time. If anything goes wrong during initialization, `Util.initialize` throws an exception.

> The semantics of Java arrays prevents this simple `Util.initialize` from modifying the argument vector. You can use another overload of `Util.initialize` to receive an argument vector with all Ice-related arguments removed.

Once you no longer need a `Communicator`, you must call `destroy` on this `Communicator`. The `destroy` method is responsible for finalizing the instance of the Ice run time embodied by this Communicator. In particular, in an Ice server, `destroy` waits for operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your application to terminate without calling `destroy` first.

The general shape of our `main` method becomes:

Java

```
public class App
{
    public static void main(String[] args)
    {
        int status = 0;
        com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.in
itialize(args);

        //
        // Register shutdown hook to destroy communicator during JVM
shutdown
        //
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
communicator.destroy(); }));

        // ...
        System.exit(status);
    }
}
```

The only pitfall with the code above is if you neglect to call `System.exit`, the application will continue to run because the Communicator started non-daemon threads.

Another way to initialize and destroy a Communicator is with a try-with-resources statement:

Java

```
public class App
{
    public static void main(String[] args)
    {
        int status = 0;
        try(com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(args))
        {
            // ...
        } // communicator is destroyed automatically here
        System.exit(status);
    }
}
```

> The `Communicator` interface implements `java.lang.AutoCloseable`: at the end of a try-with-resources statement, the communicator is closed (destroyed) automatically, without an explicit call to the `destroy` method.

If you initialize your communicator in a try-with-resources statement, you may also add a shutdown hook to destroy the communicator. `destroy` does not throw any exception and calling `destroy` multiple times is perfectly ok. A shutdown hook that calls `destroy` is useful to interrupt long-running Ice invocations when the application receives a user interrupt such as Ctrl-C.

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-Java Mapping

In this section, we present the client-side Slice-to-Java mapping. The client-side Slice-to-Java mapping defines how Slice data types are translated to Java types, and how clients invoke operations, pass parameters, and handle errors. Much of the Java mapping is intuitive. For example, Slice sequences map to Java arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in Java.

The Java API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the Java mapping, you should need no more than the Slice definition of your application and knowledge of the Java mapping rules. In particular, looking through the generated code in order to discern how to use the Java mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **Ice Packaging**
> All of the APIs for the Ice run time are nested in the `com.zeroc.Ice` package to avoid clashes with definitions for other libraries or applications. Some of the contents of the package are generated from Slice definitions; other parts of the package provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of this package throughout the remainder of the manual. For the sake of brevity, the discussions and code examples usually omit the Ice package prefix.

Topics

Java Mapping for Identifiers

A Slice identifier maps to an identical Java identifier. For example, the Slice identifier `Clock` becomes the Java identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a Java keyword or is an identifier reserved by the Ice run time (such as `checkedCast`), the corresponding Java identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.

> You should try to avoid such identifiers as much as possible.

A single Slice identifier often results in several Java identifiers. For example, for a Slice interface named `Foo`, the generated Java code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_while` and `whilePrx` (*not* `_whilePrx`), that is, the underscore prefix is applied only to those generated identifiers that actually require it.

See Also

- Lexical Rules

Java Mapping for Modules

A Slice module maps to a Java package with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions. For example:

Slice

```
// Definitions at global scope here...

module M1
{
    // Definitions for M1 here...
    module M2
    {
        // Definitions for M2 here...
    }
}

// ...

module M1    // Reopen M1
{
    // More definitions for M1 here...
}
```

This definition maps to the corresponding Java definitions:

Java

```
package M1;
// Definitions for M1 here...

package M1.M2;
// Definitions for M2 here...

package M1;
// Definitions for M1 here...
```

Note that these definitions appear in the appropriate source files; source files for definitions in module `M1` are generated in directory `M1` under neath the top-level directory, and source files for definitions for module `M2` are generated in directory `M1/M2` underneath the top-level directory. You can set the top-level output directory using the `--output-dir` option with `slice2java`.

See Also

- Modules
- Using the Slice Compilers
- Java Mapping for Identifiers

Java Mapping for Built-In Types

The Slice built-in types are mapped to Java types as follows:

Slice	Java
bool	`boolean`
byte	`byte`
short	`short`
int	`int`
long	`long`
float	`float`
double	`double`
string	`String`

Mapping of Slice built-in types to Java.

See Also

- Basic Types

Java Mapping for Enumerations

A Slice enumeration maps to the corresponding enumeration in Java. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The Java mapping for `Fruit` is shown below:

Java

```
public enum Fruit implements java.io.Serializable
{
    Apple,
    Pear,
    Orange;

    public int value();

    public static Fruit valueOf(int v);

    // ...
}
```

Given the above definitions, we can use enumerated values as follows:

Java

```
Fruit f1 = Fruit.Apple;
Fruit f2 = Fruit.Orange;

if(f1 == Fruit.Apple) // Compare with constant
{
    // ...
}

if(f1 == f2)          // Compare two enums
{
    // ...
}

switch(f2)            // Switch on enum
{
    case Fruit.Apple:
        // ...
        break;
    case Fruit.Pear:
        // ...
        break;
    case Fruit.Orange:
        // ...
        break;
}
```

The Java mapping includes two methods of interest. The `value` method returns the Slice value of an enumerator, which is not necessarily the same as its ordinal value. The `valueOf` method translates a Slice value into its corresponding enumerator, or returns `null` if no match is found. Note that the generated class contains a number of other members, which we have not shown. These members are internal to the Ice run time and you must not use them in your application code (because they may change from release to release).

In the `Fruit` definition above, the Slice value of each enumerator matches its ordinal value. This will not be true if we modify the definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The table below shows the new relationship between ordinal value and Slice value:

Enumerator	Ordinal	Slice
`Apple`	0	0
`Pear`	1	3
`Orange`	2	4

Java enumerated types inherit implicitly from `java.lang.Enum`, which defines methods such as `ordinal` and `compareTo` that operate on the *ordinal* value of an enumerator, not its Slice value.

See Also

- Enumerations

Java Mapping for Structures

On this page:

Basic Java Mapping for Structures

A Slice structure maps to a Java class with the same name. For each Slice data member, the Java class contains a corresponding public data member. For example, here is our Employee structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The Slice-to-Java compiler generates the following definition for this structure:

Java

```
public final class Employee implements java.lang.Cloneable,
java.io.Serializable
{
    public long number;
    public String firstName;
    public String lastName;

    public Employee() {}

    public Employee(long number, String firstName, String lastName)
    {
        this.number = number;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public boolean equals(java.lang.Object rhs)
    {
        // ...
    }
    public int hashCode()
    {
        // ...
    }

    public java.lang.Object clone()
    {
        java.lang.Object o;
        try
        {
            o = super.clone();
        }
        catch(java.lang.CloneNotSupportedException ex)
        {
            assert false; // impossible
        }
        return o;
    }
}
```

For each data member in the Slice definition, the Java class contains a corresponding public data member of the same name. Note that you can optionally customize the mapping for data members to use getters and setters instead.

The `equals` member function compares two structures for equality. Note that the generated class also provides the usual `hashCode` and `clone` methods. (`clone` has the default behavior of making a shallow copy.)

Java Default Constructors for Structures

Structures have a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default Java behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

Structures also have a second constructor that has one parameter for each data member. This allows you to construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members).

See Also

- Structures
- Customizing the Java Mapping

Java Mapping for Sequences

A Slice sequence maps to a Java array. This means that the Slice-to-Java compiler does not generate a separate named type for a Slice sequence.

For example:

Slice

```
sequence<Fruit> FruitPlatter;
```

This definition simply corresponds to the Java type `Fruit[]`. Naturally, because Slice sequences are mapped to Java arrays, you can take advantage of all the array functionality provided by Java, such as initialization, assignment, cloning, and the `length` member. For example:

Java

```
Fruit[] platter = { Fruit.Apple, Fruit.Pear };
assert(platter.length == 2);
```

Alternate mappings for sequence types are also possible.

See Also

- Sequences
- Customizing the Java Mapping

Java Mapping for Dictionaries

Here is the definition of our EmployeeMap once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Java mapping does not create a separate named type for this definition. Instead, the dictionary is simply an instance of the generic type `java.util.Map<K, V>`, where *K* is the mapping of the key type and *V* is the mapping of the value type. In the example above, `EmployeeMap` is mapped to the Java type `java.util.Map<Long, Employee>`. The following code demonstrates how to allocate and use an instance of `EmployeeMap`:

Java

```
java.util.Map<Long, Employee> em =
new java.util.HashMap<Long, Employee>();

Employee e = new Employee();
e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.put(e.number, e);
```

The type-safe nature of the mapping makes iterating over the dictionary quite convenient:

Java

```
em.forEach((num, employee) ->
    {

System.out.println(employee.firstName + " was employee #" + num);
    });
```

Alternate mappings for dictionary types are also possible.

See Also

- Dictionaries
- Customizing the Java Mapping

Java Mapping for Constants

Here are the sample constant definitions once more:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

Java

```
public interface AppendByDefault
{
    boolean value = true;
}

public interface LowerNibble
{
    byte value = 15;
}

public interface Advice
{
    String value = "Don't Panic!";
}

public interface TheAnswer
{
    short value = 42;
}

public interface PI
{
    double value = 3.1416;
}

public interface FavoriteFruit
{
    Fruit value = Fruit.Pear;
}
```

As you can see, each Slice constant is mapped to a Java interface with the same name as the constant. The interface contains a member named `value` that holds the value of the constant.

Slice string literals that contain non-ASCII characters or universal character names are mapped to Java string literals with universal character names. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

Java

```
public interface Egg
{
    String value = "\u0153uf";
}

public interface Heart
{
    String value = "c\u0153ur";
}

public interface Banana
{
    String value = "\ud83c\udf4c";
}
```

See Also

- Constants and Literals
- Java Mapping for Identifiers
- Java Mapping for Built-In Types

Java Mapping for Exceptions

On this page:

Java Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

Java

```
public class GenericError extends com.zeroc.Ice.UserException
{
    public GenericError()
    {
        this.reason = "";
    }

    public GenericError(Throwable cause)
    {
        super(cause);
        this.reason = "";
    }

    public GenericError(String reason)
    {
        this.reason = reason;
    }

    public GenericError(String reason, Throwable cause)
    {
        super(cause);
        this.reason = reason;
    }

    public String ice_id()
    {
```

```
        return "::M::GenericError";
    }

    public String reason;

    ...
}


public class BadTimeVal extends GenericError
{
    public BadTimeVal()
    {
        super();
    }

    public BadTimeVal(Throwable cause)
    {
        super(cause);
    }

    public BadTimeVal(String reason)
    {
        super(reason);
    }

    public BadTimeVal(String reason, Throwable cause)
    {
        super(reason, cause);
    }

    public String ice_id()
    {
        return "::M::BadTimeVal";
    }
    ...
}

public class BadZoneName extends GenericError
{
    public BadZoneName()
    {
        super();
    }

    public BadZoneName(Throwable cause)
    {
        super(cause);
    }
```

```
public BadZoneName(String reason)
{
    super(reason);
}

public BadZoneName(String reason, Throwable cause)
{
    super(reason, cause);
}

public String ice_id()
{
    return "::M::BadZoneName";
}
```

```
    ...
}
```

Each Slice exception is mapped to a Java class with the same name. For each data member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.) A JavaBean-style API is used for optional data members, and you can customize the mapping to force required members to use this same API.

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception also defines an `ice_id` method, which returns the Slice type ID of the exception.

All user exceptions are derived from the base class `UserException`. This allows you to catch all user exceptions generically by installing a handler for `UserException`. `UserException`, in turn, derives from `java.lang.Exception`.

`UserException` implements a `clone` method that is inherited by its derived exceptions, so you can make member-wise shallow copies of exceptions.

Note that the generated exception classes contain other methods that are not shown. However, those methods are internal to the Java mapping and are not meant to be called by application code.

Java Constructors for User Exceptions

Exceptions have a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default Java behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

If an exception declares or inherits any data members, the generated class provides a second constructor that accepts one parameter for each data member so that you can construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members). For a derived exception, this constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

The generated class may include an additional constructor if the exception declares or inherits any optional data members.

The Slice compiler generates overloaded versions of all constructors that accept a trailing `Throwable` argument for preserving an exception chain.

Java Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `LocalException` (which, in turn, derives indirectly from `java.lang.RuntimeException`).

`LocalException` implements a `clone` method that is inherited by its derived exceptions, so you can make member-wise shallow copies of exceptions.

Recall the inheritance diagram for user and run-time exceptions. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `UserException`
  This is the root of the inheritance tree for user exceptions.
- `TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `java.lang.Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- Java Mapping for Optional Data Members
- JavaBean Mapping
- Versioning
- Object Life Cycle

Java Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Java Classes Generated for an Interface

The compiler generates three source files for each Slice interface. In general, for an interface `<interface-name>`, the following source files are created by the compiler:

- `<interface-name>.java`
  This source file declares the `<interface-name>` Java interface, which is used in the server-side mapping.
- `<interface-name>Prx.java`
  This source file defines the proxy interface `<interface-name>Prx`.
- `_<interface-name>PrxI.java`
  This source file defines an implementation class for the interface's proxy. Applications should not use this type.

Proxy Interfaces in Java

On the client side, a Slice interface maps to a Java interface with methods that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Slice compiler generates the following definition for use by the client:

Java

```
public interface SimplePrx extends ObjectPrx
{
    void op();
    void op(java.util.Map<String, String> context);
}
```

As you can see, the compiler generates a *proxy interface* `SimplePrx`. In general, the generated name is `<interface-name>Prx`. If an interface is nested in a module `M`, the generated class is part of package `M`, so the fully-qualified name is `M.<interface-name>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server

and is known as a proxy instance. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. For the preceding example, we find that the operation `op` has been mapped to the method `op`. Also note that `op` is overloaded: the second version of `op` has a parameter `context` of type `java.util.Map<String, String>`. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the `context` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Because all the `<`*`interface-name`*`>Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. The proxy references handed out by the Ice run time are always of type `<`*`interface-name`*`>Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Interface Inheritance in Java

Inheritance relationships among Slice interfaces are maintained in the generated Java classes. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

The generated code for `CPrx` reflects the inheritance hierarchy:

Java

```
public interface CPrx extends APrx, BPrx
{
    ...
}
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

The `ObjectPrx` Interface in Java

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `ObjectPrx`. `ObjectPrx` provides a number of methods:

Java

```
public interface ObjectPrx
{
    boolean equals(java.lang.Object r);
    Identity ice_getIdentity();
    boolean ice_isA(String id);
    boolean ice_isA(String id, java.util.Map<String, String> context);
    String[] ice_ids();
    String[] ice_ids(java.util.Map<String, String> context);
    String ice_id();
    String ice_id(java.util.Map<String, String> context);
    void ice_ping();
    void ice_ping(java.util.Map<String, String> context);
    // ...
}
```

The methods behave as follows:

- **`equals`**
  This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

  ```
  module Ice
  {
      struct Identity
      {
          string name;
          string category;
      }
  }
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

Java

```
ObjectPrx o1 = ...;
ObjectPrx o2 = ...;
Identity i1 = o1.ice_getIdentity();
Identity i2 = o2.ice_getIdentity();

if(i1.equals(i2))
{
    // o1 and o2 denote the same object
}
else
{
    // o1 and o2 denote different objects

}
```

- **ice_isA**
  The ice_isA method determines whether the object denoted by the proxy supports a specific interface. The argument to ice_isA is a type ID. For example, to see whether a proxy of type ObjectPrx denotes a Printer object, we can write:

  Java

  ```
  ObjectPrx o = ...;
  if(o != null && o.ice_isA("::Printer"))
  {
      // o denotes a Printer object
  }
  else
  {
      // o denotes some other type of object

  }
  ```

  Note that we are testing whether the proxy is null before attempting to invoke the ice_isA method. This avoids getting a NullPointerException if the proxy is null.

- **ice_ids**
  The ice_ids method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- **ice_id**
  The ice_id method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type BasePrx, with a static type ID of ::Base, the return value of ice_id might be ::Base, or it might something more derived, such as ::Derived.

- **ice_ping**
  The ice_ping method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

The ice_isA, ice_ids, ice_id, and ice_ping methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in ObjectPrx, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Proxy Helper Methods in Java

For each Slice interface, the Slice-to-Java compiler generates static helper methods that support down-casting and type discovery:

Java

```
public interface SimplePrx extends com.zeroc.Ice.ObjectPrx
{
    // ...
    static SimplePrx checkedCast(ObjectPrx b);
    static SimplePrx checkedCast(ObjectPrx b, java.util.Map<String,
String> context);
    static SimplePrx checkedCast(ObjectPrx b, String facet);

    static SimplePrx checkedCast(ObjectPrx b, String facet,
java.util.Map<String, String> context);
    static SimplePrx uncheckedCast(ObjectPrx b);
    static SimplePrx uncheckedCast(ObjectPrx b, String facet);
    static String ice_staticId();
}
```

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a non-null reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference. Overloaded methods allow you to optionally specify a facet and a request context.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

Java

```
ObjectPrx obj = ...;        // Get a proxy from somewhere...

SimplePrx simple = SimplePrx.checkedCast(obj);
if(simple != null)
{
    // Object supports the Simple interface...
}
else
{
    // Object is not of type Simple...

}
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`. (This also explains the need for the helper method: the Ice run time must contact the server, so we cannot use a simple Java down-cast.)

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object

happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

Java

```
ObjectPrx obj = ...;                                  // Get proxy...
ProcessPrx process = ProcessPrx.uncheckedCast(obj);  // No worries...
process.launch(40, 60);                               // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a Java cast, the behavior is undefined.

Another method generated for every interface is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Using Proxy Methods in Java

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

Java

```
ObjectPrx proxy = communicator.stringToProxy(...);
proxy = proxy.ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to

repeat a `checkedCast` or `uncheckedCast` after using a factory method. Furthermore, the mapping generates type-specific factory methods so that no casts are necessary:

Java

```
ObjectPrx base = communicator.stringToProxy(...);
HelloPrx hello = HelloPrx.checkedCast(base);
hello = hello.ice_invocationTimeout(10000); // No cast is necessary
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type using `checkedCast` or `uncheckedCast`.

Object Identity and Proxy Comparison in Java

Proxies provide an `equals` method that compares proxies:

Java

```
interface ObjectPrx
{
    boolean equals(java.lang.Object r);
}
```

Note that proxy comparison with `equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `equals` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

Java

```
ObjectPrx p1 = ...;        // Get a proxy...
ObjectPrx p2 = ...;        // Get another proxy...

if(!p1.equals(p2))
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Util` class:

Java

```
public final class Util
{
    public static int proxyIdentityCompare(ObjectPrx lhs,
ObjectPrx rhs);
    public static int proxyIdentityAndFacetCompare(ObjectPrx lhs,
ObjectPrx rhs);
    // ...
}
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

Java

```
ObjectPrx p1 = ...;        // Get a proxy...
ObjectPrx p2 = ...;        // Get another proxy...

if(Util.proxyIdentityCompare(p1, p2) != 0)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, `-1` if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

In addition, the Java mapping provides two wrapper classes that allow you to wrap a proxy for use as the key of a hashed collection:

Java

```
public class ProxyIdentityKey
{
    public ProxyIdentityKey(ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public ObjectPrx getProxy();
}

public class ProxyIdentityFacetKey
{
    public ProxyIdentityFacetKey(ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public ObjectPrx getProxy();
}
```

The constructor caches the identity and the hash code of the passed proxy, so calls to `hashCode` and `equals` can be evaluated efficiently. The `getProxy` method returns the proxy that was passed to the constructor.

As for the comparison functions, `ProxyIdentityKey` only uses the proxy's identity, whereas `ProxyIdentityFacetKey` also includes the facet name.

Deserializing Proxies in Java

Proxy objects implement the `java.io.Serializable` interface that enables serialization of proxies to and from a byte stream. You can use the standard class `java.io.ObjectInputStream` to deserialize all Slice types *except* proxies; proxies are a special case because they must be created by a communicator.

To supply a communicator for use in deserializing proxies, an application must use the Ice-provided class `ObjectInputStream`:

Java

```
public class ObjectInputStream extends java.io.ObjectInputStream
{
    public ObjectInputStream(Communicator communicator,
java.io.InputStream stream)
        throws java.io.IOException;

    public Communicator getCommunicator();
}
```

The code shown below demonstrates how to use this class:

Java

```
Communicator communicator = ...
byte[] bytes = ... // data to be deserialized
java.io.ByteArrayInputStream byteStream =
new java.io.ByteArrayInputStream(bytes);
ObjectInputStream in = new ObjectInputStream(communicator, byteStream);
ObjectPrx proxy = (ObjectPrx)in.readObject();
```

Ice raises `java.io.IOException` if an application attempts to deserialize a proxy without supplying a communicator.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Type IDs
- Java Mapping for Operations
- Request Contexts
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

Java Mapping for Operations

On this page:

Basic Java Mapping for Operations

As we saw in the mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

Java

```
NodePrx node = ...;             // Initialize proxy
String name = node.name();      // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

Normal and `idempotent` Operations in Java

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent  string op2();
}
```

The proxy interface for this is:

Java

```
public interface ExamplePrx extends ObjectPrx
{
    String op1();
    String op2();
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

Passing Parameters in Java

In Parameters in Java

The parameter passing rules for the Java mapping are very simple: parameters are passed either by value (for simple types) or by reference (for complex types and type `string`). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy for these definitions:

Java

```
public interface ClientToServerPrx extends ObjectPrx
{
    void op1(int i, float f, boolean b, String s);
    void op2(NumberAndString ns, String[] ss, java.util.Map<Long,
String[]> st);
    void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

Java

```
ClientToServerPrx p = ...;              // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
boolean b = true;
String s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
String[] ss = { "Hello world!" };
java.util.Map<Long, String[]> st = new java.util.HashMap<Long,
String[]>();
st.put(0, ns);
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Out Parameters in Java

The mapping for an operation depends on how many values it returns, including out parameters and a non-`void` return value:

- Zero values
  The corresponding Java method returns `void`. For the purposes of this discussion, we're not interested in these operations.
- One value
  The corresponding Java method returns the mapped type, regardless of whether the Slice definition of the operation declared it as a return value or as an out parameter. Consider this example:

Slice

```
interface I
{
    string op1();
    void op2(out string name);
}
```

The mapping generates corresponding methods with identical signatures:

Java

```
interface IPrx extends ObjectPrx
{
    String op1();
    String op2();
}
```

- Multiple values
  The Slice-to-Java translator generates an extra nested class to hold the results of an operation that returns multiple values. The class is nested in the mapped interface (not the proxy interface) and has the name *Op*`Result`, where *Op* represents the name of the operation. The leading character of the class name for a "result class" is always capitalized. The values of out parameters are provided in corresponding data members of the same names. If the operation declares a return value, its value is provided in the data member named `returnValue`. If an out parameter is also named `returnValue`, the data member to hold the operation's return value is named `_returnValue` instead. The result class defines an empty constructor as well as a "one-shot" constructor that accepts and assigns a value for each of its data members. The corresponding Java method returns the result class type.

Here are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction, along with one additional operation to better demonstrate the mapping:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
    StringSeq op4(out string returnValue);
}
```

The Slice compiler generates the following code for these definitions:

Java

```
public interface ServerToClientPrx extends ObjectPrx
{
    ServerToClient.Op1Result op1();
    ServerToClient.Op2Result op2();
    ServerToClientPrx op3();
    ServerToClient.Op4Result op4();
}

public interface ServerToClient extends ...
{
    public static class Op1Result
    {
        public Op1Result() {}
        public Op1Result(int i, float f, boolean b, String s)
        {
            this.i = i;
            this.f = f;
            this.b = b;
            this.s = s;
        }
        public int i;
        public float f;
        public boolean b;
```

```
        public String s;
    }

    public static class Op2Result
    {
        public Op2Result() {}
        public Op2Result(NumberAndString ns, String[] ss,
java.util.Map<java.lang.Long, String[]> st)
        {
            this.ns = ns;
            this.ss = ss;
            this.st = st;
        }
        public NumberAndString ns;
        public String[] ss;
        public java.util.Map<java.lang.Long, String[]> st;
    }

    public static class Op4Result
    {
        public Op4Result() {}
        public Op4Result(String[] _returnValue, String returnValue)
        {
            this._returnValue = _returnValue;
            this.returnValue = returnValue;
        }
        public String[] _returnValue;
```

```
        public String returnValue;
    }
}
```

We need to point out several things here:

- Result classes are generated for `op1`, `op2` and `op4` because they return multiple values
- The result classes are generated as nested classes of interface `ServerToClient`, and **not** `ServerToClientPrx`
- `op4` declares an out parameter named `returnValue`, therefore `Op4Result` declares a data member named `_returnValue` to hold the operation's return value
- No result class is necessary for `op3` because it only returns one value; the mapped Java method declares a return type of `ServerToClientPrx` even though the Slice operation declared it as an out parameter

Null Parameters in Java

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid `NullPointerException`. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Optional Parameters in Java

The mapping uses standard Java types to encapsulate optional parameters:

- `java.util.OptionalDouble`
  The mapped type for an optional `double`.
- `java.util.OptionalInt`
  The mapped type for an optional `int`.
- `java.util.OptionalLong`
  The mapped type for an optional `long`.
- `java.util.Optional<T>`
  The mapped type for all other Slice types.

Optional return values and output parameters are mapped to instances of the above classes, depending on their types. For operations with optional in parameters, the proxy provides a set of overloaded methods that accept them as optional values, and another set of methods that accept them as required values. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params);
```

The mapping for this operation is shown below:

Java

```
java.util.OptionalInt execute(String params);
java.util.OptionalInt execute(java.util.Optional<String> params);
```

For cases where you are passing values for all of the optional in parameters, it is more efficient to use the required mapping and avoid creating temporary optional values.

A client can invoke `execute` as shown below:

Java

```
java.util.OptionalInt i;

i = proxy.execute("--file log.txt");                        // required
mapping
i = proxy.execute(java.util.Optional.of("--file log.txt")); // optional
mapping
i = proxy.execute(java.util.Optional.empty());              // params is
unset

if(i.isPresent())
{
    System.out.println("value = " + i.get());
}
```

Passing `null` where an optional value is expected is equivalent to passing an instance whose value is unset.

Java's optional classes do not consider `null` to be a legal value. Consider this example:

Slice

```
class Data
{
    ...
}

interface Repository
{
    void addOptional(optional(1) Data d);
    void addRequired(Data d);
}
```

The Ice encoding allows `null` values for class instances, so you can pass `null` to `addRequired` and the server will receive it as `null`. However, there's no way to pass an "optional value set to null" in the Java mapping. Passing `null` to `addOptional` is equivalent to passing the value of `java.util.Optional.ofNullable((T)null)`, which is equivalent to passing the value of `java.util.Optional.empty()`. In either case, the server will receive it as an optional whose value is not present.

A well-behaved program must not assume that an optional parameter always has a value. Calling `get` on an optional instance for which no

value is set raises `java.util.NoSuchElementException`.

Exception Handling in Java

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as Java exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

Java

```
ChildPrx child = ...;   // Get child proxy...

try
{
    child.askToCleanUp();
}
catch(Tantrum t)
{
    System.out.write("The child says: ");
    System.out.writeln(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

Java

```
public class Client
{
    public static void main(String[] args)
    {
        try
        {
            ChildPrx child = ...;   // Get child proxy...
            try
            {
                child.askToCleanUp();
                child.praise();     // Give positive feedback...
            }
            catch(Tantrum t)
            {
                System.out.print("The child says: ");
                System.out.println(t.reason);
                child.scold();       // Recover from error...
            }
        }
        catch(com.zeroc.Ice.LocalException e)
        {
            e.printStackTrace();
        }
    }
}
```

Exceptions and Out-Parameters

For the Java Compat mapping, the Ice run time makes no guarantees about the state of out parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

> This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

See Also

- Operations
- Java Mapping for Exceptions
- Java Mapping for Interfaces
- Java Mapping for Optional Data Members
- Collocated Invocation and Dispatch

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.

Java Mapping for Classes

On this page:

Basic Java Mapping for Classes

A Slice class is mapped to a Java class with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions). Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

The Slice compiler generates the following code for this definition:

Java

```
public class TimeOfDay extends com.zeroc.Ice.Value
{
    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second, String tz);

    public short hour;
    public short minute;
    public short second;
    public String tz;

    public TimeOfDay clone();

    public static final String ice_staticId = "::...::TimeOfDay";

    public static String ice_staticId();

    @Override
    public String ice_id();

    ...
}
```

There are a several things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `com.zeroc.Ice.Value`. This means that all classes implicitly inherit from `Value`, which is the ultimate ancestor of all classes.
2. The generated class contains a public member for each Slice data member.
3. The generated class has a constructor that takes one argument for each data member, as well as a default constructor.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice::Value` in Java

Classes implicitly inherit from a common base class, `Value`, which is mapped to `com.zeroc.Ice.Value`. `Value` is a very simple base class with just a few methods:

Java

```
public abstract class Value implements java.lang.Cloneable,
java.io.Serializable
{
    public Value clone();
    public void ice_preMarshal();
    public void ice_postUnmarshal();
    public String ice_id();
    public SlicedData ice_getSlicedData();
    ...
}
```

The `Value` methods behave as follows:

- `ice_preMarshal`
  The Ice run time invokes this method prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this method after unmarshaling an object's state. A subclass typically overrides this method when it needs to perform additional initialization using the values of its declared data members.
- `clone`
  This method returns a shallow member-wise copy of the value.
- `ice_id`
  This method returns the actual run-time type ID for a class instance. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.

Class Data Members in Java

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member. A JavaBean-style API is used for optional data members, and you can customize the mapping to force required members to use this same API.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    ["protected"] string tz;    // e.g. GMT, PST, EDT...
}
```

The Slice compiler produces the following generated code for this definition:

Java

```
public class TimeOfDay extends ...
{
    protected short hour;
    protected short minute;
    protected short second;
    protected String tz;

    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second);
    // ...
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

You can optionally customize the mapping for data members to use getters and setters instead.

Class Operations in Java

Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

Operations in classes are not mapped at all into the corresponding Java class. The generated Java class is the same whether the Slice class has operations or not.

For a class that defines or inherits operations, the Slice-to-Java compiler generates instead a separate `_<class-name>Disp` class that can be used to implement an Ice object with these operations. Let's change our example to add a class operation:

Slice

```
class FormattedTimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
    string format();
}
```

The Slice compiler generates the following code:

Java

```
public class FormattedTimeOfDay extends com.zeroc.Ice.Value
{
    // ... operation format() not mapped at all here
}

// Disp class for servant implementation
public interface _FormattedTimeOfDayDisp extends com.zeroc.Ice.Object
{
    String format(com.zeroc.Ice.Current __current);

    ...
}
```

This `Disp` class is the *skeleton class* for this Slice class. Skeleton classes are described in the Server-Side Java Mapping for Interfaces.

Value Factories in Java

Value factories allow you to create classes derived from the Java classes generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
interface Time
{
    TimeOfDay get();
}
```

The default behavior of the Ice run time will create and return an instance of the generated `TimeOfDay` class.

If you wish, you can create your own custom derived class, and tell Ice to create and return these instances instead. For example:

Java

```
public class CustomTimeOfDay extends TimeOfDay
{
    public String format() { ... prints formatted data members ... }
}
```

You then create and register a value factory for your custom class with your Ice communicator:

Java

```
Communicator communicator = ...;
communicator.getValueFactoryManager().add(type -> {
        assert(type.equals(TimeOfDay.ice_staticId()));
        return new CustomTimeOfDay();
    },
    TimeOfDay.ice_staticId());
```

Class Constructors in Java

Classes have a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default Java behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

The generated class also contains a second constructor that accepts one argument for each member of the class. This allows you to create and initialize a class in a single statement, for example:

Java

```
TimeOfDay tod = new TimeOfDay(14, 45, 00, "PST"); // 14:45pm PST
```

For derived classes, the constructor requires an argument for every member of the class, including inherited members. For example, consider the the definition from Class Inheritance once more:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}

class DateTime extends TimeOfDay
{
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
}
```

The constructors for the generated classes are as follows:

Java

```
public class TimeOfDay extends ...
{
    public TimeOfDay() {}

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // ...
}

public class DateTime extends TimeOfDay
{
    public DateTime() {}

    public DateTime(short hour, short minute, short second,
short day, short month, short year)
    {
        super(hour, minute, second);
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}
```

If you want to instantiate and initialize a `DateTime` instance, you must either use the default constructor or provide values for all of the data members of the instance, including data members of any base classes.

See Also

- Classes
- Class Inheritance
- Java Mapping for Optional Data Members
- Type IDs
- Serializable Objects in Java
- JavaBean Mapping
- The Current Object
- Dispatch Interceptors
- Value Factories

Java Mapping for Optional Data Members

The mapping for optional data members in Slice classes and exceptions uses a JavaBean-style API that provides methods to get, set, and clear a member's value, and test whether a value is set. Consider the following Slice definition:

Slice

```
class C
{
    string name;
    optional(2) string alternateName;
    optional(5) bool active;
}
```

The generated Java code provides the following API:

Java

```
public class C ...
{
    public C();
    public C(String name);
    public C(String name, String alternateName, boolean active);

    public String name;

    public String getAlternateName();
    public void setAlternateName(String v);
    public boolean hasAlternateName();
    public void clearAlternateName();
    public void optionalAlternateName(java.util.Optional<String> v);
    public java.util.Optional<String> optionalAlternateName();

    public boolean getActive();
    public void setActive(boolean v);
    public boolean isActive();
    public boolean hasActive();
    public void clearActive();
    public void optionalActive(java.util.Optional<java.lang.Boolean> v);
    public java.util.Optional<java.lang.Boolean> optionalActive();

    ...
}
```

If a class or exception declares any required data members, the generated class includes an overloaded constructor that accepts values for just the required members; optional members remain unset unless their Slice definitions specify a default value. Another overloaded constructor accepts values for all data members.

The `has` method allows you to test whether a member's value has been set, and the `clear` method removes any existing value for a member.

Calling a `get` method when the member's value has not been set raises `java.util.NoSuchElementException`.

The `optional` methods provide an alternate API that uses standard Java types to encapsulate the value:

- `java.util.OptionalDouble`
  Encapsulates a value of type `double`
- `java.util.OptionalInt`
  Encapsulates a value of type `int`
- `java.util.OptionalLong`
  Encapsulates a value of type `long`
- `java.util.Optional<T>`
  Encapsulates all other Slice types

See Also

- Optional Data Members

Serializable Objects in Java

In Java terminology, a *serializable object* typically refers to an object that implements the `java.io.Serializable` interface and therefore supports serialization to and from a byte stream. All Java classes generated from Slice definitions implement the `java.io.Serializable` interface.

In addition to serializing Slice types, applications may also need to incorporate foreign types into their Slice definitions. Ice allows you to pass Java serializable objects directly as operation parameters or as fields of another data type. For example:

Slice

```
["java:serializable:SomePackage.JavaClass"]
sequence<byte> JavaObj;
struct MyStruct
{
    int i;
    JavaObj o;
}

interface Example
{
    void op(JavaObj inObj, MyStruct s, out JavaObj outObj);
}
```

The generated code for `MyStruct` contains a member `i` of type `int` and a member `o` of type `SomePackage.JavaClass`:

Java

```
public final class MyStruct implements java.lang.Cloneable
{
    public int i;
    public SomePackage.JavaClass o;

    // ...
}
```

Similarly, the signature for `op` has parameters of type `JavaClass` and `MyStruct` for the in-parameters and returns `JavaClass`:

Java

```
SomePackage.JavaClass op(SomePackage.JavaClass inObj, MyStruct s);
```

Of course, your client and server code must have an implementation of `JavaClass` that derives from `java.io.Serializable`:

Java

```
package SomePackage;

public class JavaClass implements java.io.Serializable
{
    // ...
}
```

You can implement this class in any way you see fit — the Ice run time does not place any other requirements on the implementation.

See Also

- Serializable Objects

Customizing the Java Mapping

You can customize the code that the Slice-to-Java compiler produces by annotating your Slice definitions with metadata. This section describes how metadata influences several aspects of the generated Java code.

On this page:

Java Packages

By default, the scope of a Slice definition determines the package of its mapped Java construct. A Slice type defined in a module hierarchy is mapped to a type residing in the equivalent Java package.

There are times when applications require greater control over the packaging of generated Java classes. For instance, a company may have software development guidelines that require all Java classes to reside in a designated package. One way to satisfy this requirement is to modify the Slice module hierarchy so that the generated code uses the required package by default. In the example below, we have enclosed the original definition of `Workflow::Document` in the modules `com::acme` so that the compiler will create the class in the `com.acme` package:

Slice

```
module com
{
    module acme
    {
        module Workflow
        {
            class Document
            {
                // ...
            }
        }
    }
}
```

There are two problems with this workaround:

1. It incorporates the requirements of an implementation language into the application's interface specification.
2. Developers using other languages, such as C++, are also affected.

The Slice-to-Java compiler provides a better way to control the packages of generated code through the use of metadata. The example above can be converted as follows:

Slice

```
["java:package:com.acme"]
module Workflow
{
    class Document
    {
        // ...
    }
}
```

The metadata directive `java:package:com.acme` instructs the compiler to generate all of the classes resulting from definitions in module `Workflow` into the Java package `com.acme`. The net effect is the same: the class for `Document` is generated in the package `com.acme.Workflow`. However, we have addressed the two shortcomings of the first solution by reducing our impact on the interface specification: the Slice-to-Java compiler recognizes the package metadata directive and modifies its actions accordingly, whereas the compilers for other language mappings simply ignore it.

> The `java:package` directive can also be applied as global metadata, in which case it serves as the default directive unless overridden by module metadata.

Java Package Configuration Properties

Using metadata to alter the default package of generated classes has ramifications for the Ice run time when unmarshaling exceptions and concrete class types. The Ice run time dynamically loads generated classes by translating their Slice type IDs into Java class names. For example, the Ice run time translates the Slice type ID `::Workflow::Document` into the class name `Workflow.Document`.

However, when the generated classes are placed in a user-specified package, the Ice run time can no longer rely on the direct translation of a Slice type ID into a Java class name, and therefore requires additional configuration so that it can successfully locate the generated classes. Two configuration properties are supported:

- `Ice.Package.`*`Module=package`*
  Associates a top-level Slice module with the package in which it was generated.

  > Only top-level module names are allowed; the semantics of global metadata prevent a nested module from being generated into a different package than its enclosing module.

- `Ice.Default.Package=package`
  Specifies a default package to use if other attempts to load a class have failed.

The behavior of the Ice run time when unmarshaling an exception or concrete class is described below:

1. Translate the Slice type ID into a Java class name and attempt to load the class.
2. If that fails, extract the top-level module from the type ID and check for an `Ice.Package` property with a matching module name. If found, prepend the specified package to the class name and try to load the class again.
3. If that fails, check for the presence of `Ice.Default.Package`. If found, prepend the specified package to the class name and try to load the class again.
4. If the class still cannot be loaded, the instance may be sliced.

Continuing our example from the previous section, we can define the following property:

```
Ice.Package.Workflow=com.acme
```

Alternatively, we could achieve the same result with this property:

```
Ice.Default.Package=com.acme
```

Custom Types in Java

One of the more powerful applications of metadata is the ability to tailor the Java mapping for sequence and dictionary types to match the needs of your application.

Custom Type Metadata in Java

The metadata for specifying a custom type has the following format:

```
java:type:instance-type[:formal-type]
```

The formal type is optional; the compiler uses a default value if one is not defined. The instance type must satisfy an is-A relationship with the formal type: either the same class is specified for both types, or the instance type must be derived from the formal type.

The Slice-to-Java compiler generates code that uses the formal type for all occurrences of the modified Slice definition except when the generated code must instantiate the type, in which case the compiler uses the instance type instead.

The compiler performs no validation on your custom types. Misspellings and other errors will not be apparent until you compile the generated code.

Defining a Custom Sequence Type in Java

Although the default mapping of a sequence type to a native Java array is efficient and typesafe, it is not always the most convenient representation of your data. To use a different representation, specify the type information in a metadata directive, as shown in the following example:

Slice

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

It is your responsibility to use a type parameter for the Java class (`String` in the example above) that is the correct mapping for the sequence's element type.

The compiler requires the formal type to implement `java.util.List<`*E*`>`, where *E* is the Java mapping of the element type. If you do not specify a formal type, the compiler uses `java.util.List<`*E*`>` by default.

Note that extra care must be taken when defining custom types that contain nested generic types, such as a custom sequence whose element type is also a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Defining a Custom Dictionary Type in Java

The default instance type for a dictionary is `java.util.HashMap<`*K*`, `*V*`>`, where *K* is the Java mapping of the key type and *V* is the Java mapping of the value type. If the semantics of a `HashMap` are not suitable for your application, you can specify an alternate type using metadata as shown in the example below:

Slice

```
["java:type:java.util.TreeMap<String, String>"]
dictionary<string, string> StringMap;
```

It is your responsibility to use type parameters for the Java class (`String` in the example above) that are the correct mappings for the dictionary's key and value types.

The compiler requires the formal type to implement `java.util.Map<K, V>`. If you do not specify a formal type, the compiler uses this type by default.

Note that extra care must be taken when defining dictionary types that contain nested generic types, such as a dictionary whose element type is a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Using Custom Type Metadata in Java

You can define custom type metadata in a variety of situations. The simplest scenario is specifying the metadata at the point of definition:

Slice

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

Defined in this manner, the Slice-to-Java compiler uses `java.util.List<String>` (the default formal type) for all occurrences of `String List`, and `java.util.LinkedList<String>` when it needs to instantiate `StringList`.

You may also specify a custom type more selectively by defining metadata for a data member, parameter or return value. For instance, the mapping for the original Slice definition might be sufficient in most situations, but a different mapping is more convenient in particular cases. The example below demonstrates how to override the sequence mapping for the data member of a structure as well as for several operations:

Slice

```
sequence<string> StringSeq;

struct S
{
    ["java:type:java.util.LinkedList<String>"] StringSeq seq;
}

interface I
{
    ["java:type:java.util.ArrayList<String>"] StringSeq
    modifiedReturnValue();

    void modifiedInParam(["java:type:java.util.ArrayList<String>"] Stri
ngSeq seq);

    void modifiedOutParam(out ["java:type:java.util.ArrayList<String>"]
StringSeq seq);
}
```

As you might expect, modifying the mapping for an operation's parameters or return value may require the application to manually convert values from the original mapping to the modified mapping. For example, suppose we want to invoke the `modifiedInParam` operation. The signature of its proxy operation is shown below:

Java

```
void modifiedInParam(java.util.List<String> seq)
```

The metadata changes the mapping of the `seq` parameter to `java.util.List`, which is the default formal type. If a caller has a `StringSeq` value in the original mapping, it must convert the array as shown in the following example:

Java

```
String[] seq = new String[2];
seq[0] = "hi";
seq[1] = "there";
IPrx proxy = ...;
proxy.modifiedInParam(java.util.Arrays.asList(seq));
```

Although we specified the instance type `java.util.ArrayList<String>` for the parameter, we are still able to pass the result of `asList` because its return type (`java.util.List<String>`) is compatible with the parameter's formal type declared by the proxy method. In the case of an operation parameter, the instance type is only relevant to a servant implementation, which may need to make assumptions about the actual type of the parameter.

Buffer Types in Java

You can annotate sequences of certain primitive types with the `java:buffer` metadata tag to change the mapping to use subclasses of `java.nio.Buffer`. This mapping provides several benefits:

- You can pass a buffer to a Slice API instead of creating and filling a temporary array
- If you need to pass a portion of an existing array, you can wrap it with a buffer and avoid an extra copy
- Receiving buffers during a Slice operation also avoids copying by directly referencing the data in Ice's unmarshaling buffer

> To use buffers safely, applications must disable caching by setting `Ice.CacheMessageBuffers` to zero.

The following table lists each supported Slice primitive type with its corresponding mapped class:

Primitive	Mapping
`byte`	`java.nio.ByteBuffer`
`short`	`java.nio.ShortBuffer`
`int`	`java.nio.IntBuffer`
`long`	`java.nio.LongBuffer`
`float`	`java.nio.FloatBuffer`
`double`	`java.nio.DoubleBuffer`

The `java:buffer` tag can be applied to the initial definition of a sequence, in which case the mapping uses the buffer type for all occurrences of that sequence type:

Slice

```
["java:buffer"] sequence<int> Values;

struct Observation
{
    int x;
    int y;
    Values measurements;
}
```

We can construct an `Observation` as follows:

Java

```
Observation obs = new Observation();
obs.x = 5;
obs.y = 9;
obs.measurements = java.nio.IntBuffer.allocate(10);
for(int i = 0; i < obs.measurements.capacity(); ++i)
{
    obs.measurements.put(i, ...);

}
```

The `java:buffer` tag can also be applied in more limited situations to override a sequence's normal mapping:

Slice

```
sequence<byte> ByteSeq; // Maps to byte[]

struct Page
{
    int offset;
    ["java:buffer"] ByteSeq data; // Maps to java.nio.ByteBuffer
}

interface Decoder
{
    ["java:buffer"] ByteSeq decode(ByteSeq data);
}
```

In this example, `ByteSeq` maps by default to a `byte` array, but we've overridden the mapping to use a buffer when this type is used as a data member in `Page` and as the return value of the `decode` operation; the input parameter to `decode` uses the default array mapping.

JavaBean Mapping

The Java mapping optionally generates JavaBean-style methods for the data members of class, structure, and exception types.

JavaBean Generated Methods

For each data member *val* of type *T*, the mapping generates the following methods:

Java

```
public T getVal();
public void setVal(T v);
```

The mapping generates an additional method if *T* is the `bool` type:

Java

```
public boolean isVal();
```

Finally, if *T* is a sequence type with an element type *E*, two methods are generated to provide direct access to elements:

Java

```
public E getVal(int index);
public void setVal(int index, E v);
```

Note that these element methods are only generated for sequence types that use the default mapping.

The Slice-to-Java compiler considers it a fatal error for a JavaBean method of a class data member to conflict with a declared operation of the class. In this situation, you must rename the operation or the data member, or disable the generation of JavaBean methods for the data member in question.

JavaBean Metadata

The JavaBean methods are generated for a data member when the member or its enclosing type is annotated with the `java:getset` meta data. The following example demonstrates both styles of usage:

Slice

```
sequence<int> IntSeq;

class C
{
    ["java:getset"] int i;
    double d;
}

["java:getset"]
struct S
{
    bool b;
    string str;
}

["java:getset"]
exception E
{
    IntSeq seq;
}
```

JavaBean methods are generated for all members of struct `S` and exception `E`, but for only one member of class `C`. Relevant portions of the generated code are shown below:

Java

```
public class C extends ...
{
    ...

    public int i;

    public int getI()
    {
        return i;
    }

    public void setI(int _i)
    {
        i = _i;
    }

    public double d;
}

public final class S implements java.lang.Cloneable
{
```

```
    public boolean b;

    public boolean getB()
    {
        return b;
    }

    public void setB(boolean _b)
    {
        b = _b;
    }

    public boolean isB()
    {
        return b;
    }

    public String str;

    public String getStr()
    {
        return str;
    }

    public void setStr(String _str)
    {
        str = _str;
    }

    ...
}

public class E extends UserException
{
    ...

    public int[] seq;

    public int[] getSeq()
    {
        return seq;
    }

    public void setSeq(int[] _seq)
    {
        seq = _seq;
    }

    public int getSeq(int _index)
    {
```

```
        return seq[_index];
    }

    public void setSeq(int _index, int _val)
    {
        seq[_index] = _val;
    }
```

```
    ...
}
```

Overriding `serialVersionUID`

The Slice-to-Java compiler computes a default value for the `serialVersionUID` member of Slice classes, exceptions and structures. If you prefer, you can override this value using the `java:serialVersionUID` metadata, as shown below:

Slice

```
["java:serialVersionUID:571254925"]
struct Identity
{
    ...
}
```

The specified value will be used in place of the default value in the generated code:

Java

```
public class Identity
{
    ...

    public static final long serialVersionUID = 571254925L;
}
```

By using this metadata, the application assumes responsibility for updating the UID whenever changes to the Slice definition affect the serializable state of the type.

See Also

- Metadata
- Java Mapping for Modules
- Java Mapping for Operations
- Class Inheritance Semantics

Asynchronous Method Invocation (AMI) in Java

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

Basic Asynchronous API in Java

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in Java

In addition to the synchronous proxy methods, `slice2java` generates the following asynchronous proxy methods:

Java

```
public interface EmployeesPrx extends com.zeroc.Ice.ObjectPrx
{
    // ...

    java.util.concurrent.CompletableFuture<java.lang.String>
getNameAsync(int number);
    java.util.concurrent.CompletableFuture<java.lang.String>
getNameAsync(int number, java.util.Map<String, String> context);
}
```

As you can see, the `getName` Slice operation generates a `getNameAsync` method, along with an overload so that you can pass a per-invocation context.

The `getNameAsync` method sends (or queues) an invocation of `getName`. This method does not block the calling thread. It returns an instance of `java.util.concurrent.CompletableFuture` that you can use in a number of ways, including blocking to obtain the result, configuring an action to be executed when the result becomes available, and canceling the invocation.

Here's an example that calls `getNameAsync`:

Java

```
EmployeesPrx e = ...;
java.util.concurrent.CompletableFuture<String> f = e.getNameAsync(99);

// Continue to do other things here...

String name = f.join();
```

Because `getNameAsync` does not block, the calling thread can do other things while the operation is in progress.

An asynchronous proxy method uses the same parameter mapping as for synchronous operations; the only difference is that the result (if any) is returned in a `CompletableFuture`. An operation that returns no values maps to an asynchronous proxy method that returns `CompletableFuture<Void>`. For example, consider the following operation:

Slice

```
interface Example
{
    double op(int inp1, string inp2, out bool outp1, out long outp2);
}
```

The generated code looks like this:

Java

```
public interface Example
{
    public static class OpResult
    {
        public OpResult();
        public OpResult(double returnValue, boolean outp1, long outp2);

        public double returnValue;
        public boolean outp1;
        public long outp2;
    }

    ...
}

public interface ExamplePrx extends com.zeroc.Ice.ObjectPrx
{
    java.util.concurrent.CompletableFuture<Example.OpResult> opAsync(int
inp1, String inp2);
    java.util.concurrent.CompletableFuture<Example.OpResult> opAsync(int
inp1, String inp2, java.util.Map<String, String> context);

    ...
}
```

Now let's call `whenComplete` to demonstrate one way of asynchronously executing an action when the invocation completes:

Java

```
ExamplePrx e = ...;
e.opAsync(5, "demo").whenComplete((result, ex) ->
    {
        if(ex != null)
        {
            // handle exception...
        }
        else
        {
            System.out.println("returnValue = " + result.returnValue);
            System.out.println("outp1 = " + result.outp1);
            System.out.println("outp2 = " + result.outp2);
        }
    });
```

Asynchronous Exception Semantics in Java

If an invocation raises an exception, the exception can be obtained from the future in several ways:

- Call `get` on the future; `get` raises `CompletionException` with the actual exception available via `getCause()`
- Call `join` on the future; `join` raises `ExecutionException` with the actual exception available via `getCause()`
- Use chaining methods such as `exceptionally`, `handle` or `whenComplete` to execute custom actions

The exception is provided by the future, even if the actual error condition for the exception was encountered during the call to the `opAsync` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that handles the future (instead of being present twice, once where the `opAsync` method is called, and again where the future is handled).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

`InvocationFuture` Class in Java

The `CompletableFuture` object that is returned by asynchronous proxy methods can be down-cast to `InvocationFuture` when an application requires more control over an invocation:

Java

```
package com.zeroc.Ice;

public class InvocationFuture<T> extends ...
{
    public Communicator getCommunicator();
    public Connection getConnection();
    public ObjectPrx getProxy();
    public String getOperation();

    public void waitForCompleted();

    public boolean isSent();
    public void waitForSent();
    public boolean sentSynchronously();

    public CompletableFuture<Boolean>
whenSent(java.util.function.BiConsumer<Boolean, ? super Throwable>
action);
    public abstract CompletableFuture<Boolean>
whenSentAsync(java.util.function.BiConsumer<Boolean, ? super Throwable>
action);
    public abstract CompletableFuture<Boolean>
whenSentAsync(java.util.function.BiConsumer<Boolean, ? super Throwable>
action, Executor executor);
}
```

The methods have the following semantics:

- `Communicator getCommunicator()`
  This method returns the communicator that sent the invocation.

- `Connection getConnection()`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `getConnection` method only returns a non-nil value when the future is obtained by calling `flushBatchRequestsAsync` on a `Connection` object.

- `ObjectPrx getProxy()`
  This method returns the proxy that was used to call the asynchronous proxy method, or nil if the future was not obtained via an asynchronous proxy invocation.

- `String getOperation()`
  This method returns the name of the operation.

- `void waitForCompleted()`
  This method blocks the caller until the result of an invocation becomes available. Upon return, the standard method `CompletableFuture.isDone()` returns true.

- `boolean isSent()`
  When you call an asynchronous proxy method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.

- `void waitForSent()`
  This method blocks the calling thread until a request has been written to the client-side transport, or an exception occurs. After `waitForSent` returns, `isSent` returns true if the request was successfully written to the client-side transport, or false if an exception occurred. In the case of a failure, you can obtain the exception using the standard `CompletableFuture` methods.

- `boolean sentSynchronously()`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

- `CompletableFuture<Boolean> whenSent(BiConsumer<Boolean, ? super Throwable> action)`
  Configures an action to be executed when the request has been successfully written to the client-side transport. The arguments to the action are a boolean indicating whether the request was sent synchronously (see `sentSynchronously` above) and a `Throwable`. The exception argument will be null if the request was sent successfully. The returned stage is completed when the action returns. If the supplied action itself encounters an exception, then the returned stage exceptionally completes with this exception unless this stage also completed exceptionally. If the invocation is already sent at the time `whenSent` is called, the callback method is invoked recursively from the calling thread. Otherwise, the callback method is invoked by an Ice thread (or by a dispatcher if one is configured).

- `CompletableFuture<Boolean> whenSentAsync(BiConsumer<Boolean, ? super Throwable> action)`
  Behaves like `whenSent` except the given action is executed asynchronously using this stage's default asynchronous execution facility.

- `CompletableFuture<Boolean> whenSentAsync(BiConsumer<Boolean, ? super Throwable> action, Executor executor)`
  Behaves like `whenSent` except the given action is executed using the supplied executor.

Due to limitations in Java's generic type system, a regular down-cast from `CompletableFuture<T>` to `InvocationFuture<T>` would cause a compiler error, therefore Ice provides a helper method to perform the conversion for you:

Java

```
package com.zeroc.Ice;

public class Util
{
    static public InvocationFuture
getInvocationFuture(java.util.concurrent.CompletableFuture f);

    ...
}
```

See below for sample code that uses this method.

Polling for Completion in Java

The `InvocationFuture` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

Java

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;
while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

Java

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;

LinkedList<InvocationFuture<Void>> results =
new LinkedList<InvocationFuture<Void>>();
int numRequests = 5;

while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    CompletableFuture<Void> f = ft.sendAsync(offset, bs);
    offset += bs.length;

    // Wait until this request has been passed to the transport.
    InvocationFuture<Void> i = Util.getInvocationFuture(f);
    i.waitForSent();
    results.add(i);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.size() > numRequests)
    {
        i = results.getFirst();
        results.removeFirst();
        i.join();
    }
}

// Wait for any remaining requests to complete.
while(results.size() > 0)
{
    InvocationFuture<Void> i = results.getFirst();
    results.removeFirst();
    i.join();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger

or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Asynchronous Oneway Invocations in Java

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous proxy method on a oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

The future returned for a oneway invocation completes as soon as the request is successfully written to the client-side transport. The future completes exceptionally if an error occurs before the request is successfully written.

Flow Control in Java

Asynchronous method invocations never block the thread that calls the asynchronous proxy method. The Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `InvocationFuture.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `InvocationFuture.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The `InvocationFuture` class provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport:

Java

```
ExamplePrx proxy = ...;

CompletableFuture<Result> f = proxy.doSomethingAsync();
InvocationFuture<Result> i = Util.getInvocationFuture(f);
i.whenSent((sentSynchronously, ex) ->
    {
        if(ex != null)
        {
            // handle errors...
        }
        else
        {
            // this request was sent, send another!
        }
    });
```

The `whenSent` method has the following semantics:

- If the Ice run time was able to pass the entire request to the local transport immediately, the action will be invoked from the current thread and the `sentSynchronously` argument will be true.
- If Ice wasn't able to write the entire request without blocking, the action will eventually be invoked from an Ice thread pool thread and the `sentSynchronously` argument will be false.

If you need more control over the execution environment of your action, you can use one of the `whenSentAsync` methods instead. The `sentSynchronously` argument still behaves as described above, but your executor's implementation will determine the threading behavior.

Asynchronous Batch Requests in Java

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous proxy method on a batch oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

The future returned for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The future completes exceptionally if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

The proxy method `ice_flushBatchRequestsAsync` flushes any batch requests queued by that proxy. In addition, similar methods are available on the communicator and the `Connection` object that is returned by `InvocationFuture.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Canceling Asynchronous Requests in Java

`CompletableFuture` provides a `cancel` method that you can call to cancel an invocation. If the future hasn't already completed either successfully or exceptionally, cancelling the future causes it to complete with an instance of `java.util.concurrent.CancellationException`.

> Cancellation prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. Cancellation is a local operation and has no effect on the server.

Concurrency Semantics for AMI in Java

For the `CompletableFuture` returned by an asynchronous proxy method, the Ice run time invokes `complete` or `completeExceptionally` from an Ice thread pool thread. The thread in which your action executes depends on the completion status of the future and the manner in which you registered the action. Here are some examples:

- Suppose you configure an action using `whenComplete`. If the future is already complete at the time you call `whenComplete`, the action will execute immediately in the calling thread. If the future is not yet complete when you call `whenComplete`, the action will eventually execute in an Ice thread pool thread.
- Now suppose you configure an action using one of the `whenCompleteAsync` methods. Regardless of the thread in which Ice completes the future, your executor's implementation will determine the thread context in which the action is invoked. The Ice thread pool can be used as an executor; you can obtain the executor by calling the `ice_executor` proxy method and passing it to `whenCompleteAsync`. With the Ice thread pool executor, the action is always queued to be executed by the Ice thread pool. If a dispatcher is configured, the action will be passed to the configured dispatcher by the Ice thread pool thread that dequeues it, otherwise the action will be executed by the Ice thread pool thread that dequeues it.

Refer to the flow control discussion for information about the concurrency semantics of the flow control methods.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch

Using the Slice Compiler for Java

Redirection Notice
This page will redirect to slice2java Command-Line Options in about one second.

slice2java Command-Line Options

The Slice-to-Java compiler, `slice2java`, offers the command-line options described below in addition to the standard options. Note that some of these options are only applicable for the Java Compat mapping, meaning they must be used together with the `--compat` option.

- `--compat`
  Generate code for the Java Compat mapping. When this option is specified, the macro `__SLICE2JAVA_COMPAT__` is defined during the compilation. If this option is not specified, the default behavior is to generate code for the Java mapping.
- `--tie`
  Generate tie classes. (Java Compat only)
- `--impl`
  Generate sample implementation files. This option will not overwrite an existing file.
- `--impl-tie`
  Generate sample implementation files using tie classes. This option will not overwrite an existing file. (Java Compat only)
- `--checksum` *`CLASS`*
  Generate checksums for Slice definitions into the class *`CLASS`*. The given class name may optionally contain a package specifier. The generated class contains checksums for all of the Slice files being translated by this invocation of the compiler. For example, the command below causes `slice2java` to generate the file `Checksums.java` containing the checksums for the Slice definitions in `File1.ice` and `File2.ice`:

  ```
  slice2java --checksum Checksums File1.ice File2.ice
  ```

- `--meta` *`META`*
  Define the global metadata directive *`META`*. Using this option is equivalent to defining the global metadata *`META`* in each named Slice file, as well as in any file included by a named Slice file. Global metadata specified with `--meta` overrides any corresponding global metadata directive in the files being compiled.
- `--list-generated`
  Emit a list of generated files in XML format.

See Also

- Using the Slice Compilers
- Using Slice Checksums in Java
- Tie Classes in Java Compat

Using Slice Checksums in Java

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2java`, the `--checksum` option causes the compiler to generate a new Java class that adds checksums to a static map member. Assuming we supplied the option `--checksum Checksums` to `slice2java`, the generated class `Checksums.java` looks like this:

Java

```
public class Checksums
{
    public static java.util.Map<String, String> checksums;
}
```

The read-only map `checksums` is initialized automatically prior to first use; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the dictionaries using the `equals` method. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

Java

```
java.util.Map<String, String> serverChecksums = ...
Checksums.checksums.forEach((id, checksum) ->
    {
        String serverChecksum = serverChecksums.get(id);
        if(serverChecksum == null)
        {
            // No match found for type id!
        }
        else if(!checksum.equals(serverChecksum))
        {
            // Checksum mismatch!
        }
    });
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums

Example of a File System Client in Java

This page presents the source code for a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The Java code hardly differs from the code you would write for an ordinary Java program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Java object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side Java mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

Java

```
import Filesystem.*;

public class Client
{
```

```
    // Recursively print the contents of directory "dir" in
    // tree fashion.  For files, show the contents of each file.
    // The "depth" parameter is the current nesting level
    // (for indentation).

    static void listRecursive(DirectoryPrx dir, int depth)
    {
        char[] indentCh = new char[++depth];
        java.util.Arrays.fill(indentCh, '\t');
        String indent = new String(indentCh);

        NodePrx[] contents = dir.list();

        for(int i = 0; i < contents.length; ++i)
        {
            DirectoryPrx subdir = DirectoryPrx.checkedCast(contents[i]);
            FilePrx file = FilePrx.uncheckedCast(contents[i]);
            System.out.println(indent + contents[i].name() +
(subdir != null ? " (directory):" : " (file):"));
            if(subdir != null)
            {
                listRecursive(subdir, depth);
            }
            else
            {
                String[] text = file.read();
                for(int j = 0; j < text.length; ++j)
                {
                    System.out.println(indent + "\t" + text[j]);
                }
            }
        }
    }

    public static void main(String[] args) throws Exception
    {
        try(com.zeroc.Ice.Communicator ic =
com.zeroc.Ice.Util.initialize(args))
        {
            //
            // Create a proxy for the root directory
            //
            com.zeroc.Ice.ObjectPrx base
= ic.stringToProxy("RootDir:default -p 10000");
            if(base == null)
            {
                throw new RuntimeException("Cannot create proxy");
            }

            //
```

```
        // Down-cast the proxy to a Directory proxy
        //
        DirectoryPrx rootDir = DirectoryPrx.checkedCast(base);
        if(rootDir == null)
        {
            throw new RuntimeException("Invalid proxy");
        }

        //
        // Recursively list the contents of the root directory
        //
        System.out.println("Contents of root directory:");
        listRecursive(rootDir, 0);
    }
```

```
        }
    }
```

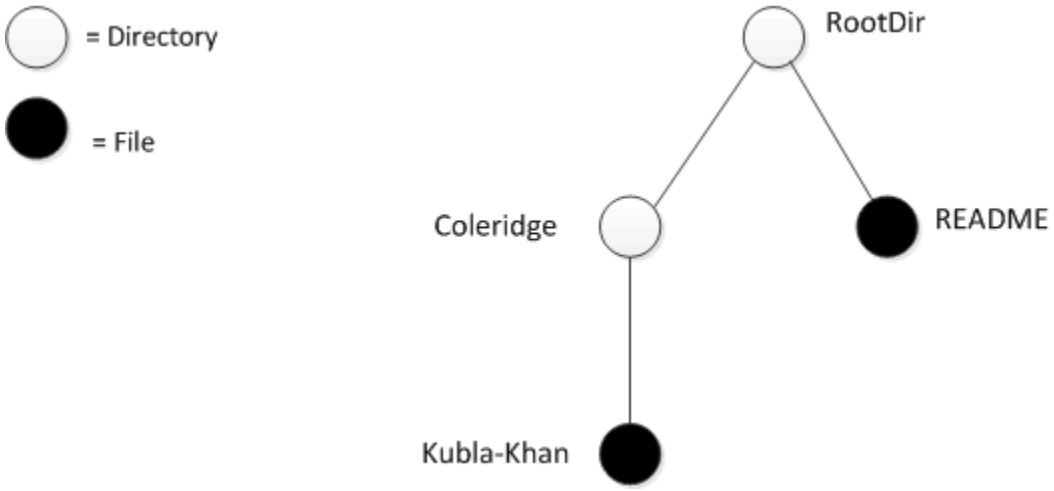
After importing the `Filesystem` package, the `Client` class defines two methods: `listRecursive`, which is a helper function to print the contents of the file system, and `main`, which is the main program. Let us look at `main` first:

1. The structure of the code in `main` follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Example of a File System Server in Java
- Object Life Cycle
- IceGrid

Server-Side Slice-to-Java Mapping

The mapping for Slice data types to Java is identical on the client side and server side. This means that everything in Client-Side Slice-to-Java Mapping also applies to the server side. However, for the server side, there are a few additional things you need to know — specifically how to:

- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time

Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Although the examples we present are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Java mapping rules beyond those we described here.

Topics

Server-Side Java Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing member functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Types in Java

On the client side, interfaces map to proxy types. On the server side, interfaces map to *skeleton* types. A skeleton is a class or interface that defines a method for each operation on the corresponding Slice interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The Slice compiler generates the following definition for this interface:

Java

```
package Filesystem;

public interface Node extends com.zeroc.Ice.Object
{
    static final String ice_staticId = "::Filesystem::Node";

    String name(com.zeroc.Ice.Current __current);

    // Mapping-internal code here...
}
```

There are two important points here:

- As for the client side, Slice modules are mapped to Java packages with the same name, so the skeleton class definitions are part of the `Filesystem` package.
- For each Slice interface <*`interface-name`*>, the compiler generates a Java interface of the same name that extends `com.zeroc.Ice.Object`, defines a method for each operation in the Slice interface, and defines the constant `ice_staticId` with the corresponding Slice type ID. This interface supplies the skeleton code; your servant must implement this interface.

`Object` Base Interface for Java Servants

`Object` is mapped to the `com.zeroc.Ice.Object` interface in Java:

Java

```
package com.zeroc.Ice;

public interface Object
{
    public class Ice_invokeResult
    {
        ...
    }

    default boolean ice_isA(String s, Current current) { ... }
    default void ice_ping(Current current) { ... }
    default String[] ice_ids(Current current) { ... }
    default String ice_id(Current current) { ... }

    public static String ice_staticId() { ... }

    default CompletionStage<OutputStream> ice_dispatch(Request request)
throws UserException { ... }

    ...
}
```

The methods of `Object` behave as follows:

- `ice_isA`
  This method returns `true` if target object implements the given type ID, and `false` otherwise.
- `ice_ping`
  `ice_ping` provides a basic reachability test for the servant.
- `ice_ids`
  This method returns a string sequence representing all of the type IDs implemented by this servant, including `::Ice::Object`.
- `ice_id`
  This method returns the type ID of the most-derived interface implemented by this servant.
- `ice_staticId`
  This static method returns the type ID of the target interface: `::Ice::Object` when called on `com.zeroc.Ice.Object`.
- `ice_dispatch`
  This method dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.

The nested class `Ice_invokeResult` supplies the result for calls to `ice_invoke` and `ice_invokeAsync` on proxies.

Servant Classes in Java

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton type. For example, to create a servant for the `Node` interface, you could write:

Java

```
package Filesystem;

public final class NodeI implements Node
{
    public NodeI(String name)
    {
        _name = name;
    }

    @Override
    public String name(com.zeroc.Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.)

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` function returns its value.)

Normal and `idempotent` Operations in Java

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

Slice

```
interface Example
{
                void   normalOp();
    idempotent void   idempotentOp();
    idempotent string readonlyOp();
}
```

The methods for this interface look like this:

Java

```
void normalOp(Current current);
void idempotentOp(Current current);
String readonlyOp(Current current);
```

Note that the signatures of the member functions are unaffected by the `idempotent` qualifier.

See Also

- Slice for a Simple File System
- Java Mapping for Interfaces
- Parameter Passing in Java
- Raising Exceptions in Java
- The Current Object

Parameter Passing in Java

Parameter passing on the server side follows the rules for the client side. Additionally, every operation receives a trailing parameter of type `Current`. For example, the `name` operation of the `Node` interface has no parameters, but the corresponding `name` method of the servant interface has a single parameter of type `Current`. We will ignore this parameter for now.

> The parameter-passing rules change somewhat when using the asynchronous mapping.

On this page:

Server-Side Mapping for Parameters in Java

The servant mapping for operations is consistent with the proxy mapping. To illustrate the rules for the Java mapping, consider the following interface:

Slice

```
module M
{
    interface Example
    {
        string op1();
        void op2(out string sout);
        string op3(string sin, out string sout);

        optional(1) string op4();
        void op5(out optional(1) string sout);
        optional(1) string op6(out optional(2) string sout);
    }
}
```

The generated skeleton interface looks like this:

Java

```
public interface Example extends com.zeroc.Ice.Object
{
    public static class Op3Result
    {
        public Op3Result();
        public Op3Result(String returnValue, String sout);

        public String returnValue;
        public String sout;
    }

    public static class Op6Result
    {
        public Op6Result();
        public Op6Result(java.util.Optional<java.lang.String>
returnValue,
                         java.util.Optional<java.lang.String> sout);
        public Op6Result(java.lang.String returnValue, java.lang.String
sout);

        public java.util.Optional<java.lang.String> returnValue;
        public java.util.Optional<java.lang.String> sout;
    }

    String op1(com.zeroc.Ice.Current current);
    String op2(com.zeroc.Ice.Current current);
    Example.Op3Result op3(String sin, com.zeroc.Ice.Current current);
    java.util.Optional<java.lang.String> op4(com.zeroc.Ice.Current
current);
    java.util.Optional<java.lang.String> op5(com.zeroc.Ice.Current
current);
    Example.Op6Result op6(com.zeroc.Ice.Current current);
}
```

You'll notice that `op1` and `op2` have same signature because the mapping rules state that an operation returning a single value shall use that type as its return value, regardless of whether the Slice operation declared it as the return type or as an out parameter. (The same is true for `op4` and `op5`.) The proxy and servant methods also share the `Result` types that are generated when an operation returns more than one value, as shown above for `op3` and `op6`. When at least one of the return value and `out` parameters is optional, the `Result` type provides two overload constructors that takes the return value followed by all `out` parameters: one with the `java.util.Optional` types, and one without `java.util.Optional` types; with the latter, a null value for an optional parameter is interpreted as meaning "not set".

The only difference between the client and server sides is the type of the extra trailing parameter.

Thread-Safe Marshaling in Java

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For Java applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following servant implementation:

Java

```
public class GridI implements Grid
{
    GridI()
    {
        _grid = // ...
    }

    public int[][] getGrid(Current current)
    {
        return _grid;
    }

    public void setValue(int x, int y, int val, Current current)
    {
        _grid[x][y] = val;
    }

    private int[][] _grid;
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned array in preparation to send a reply message, it is possible for another thread to dispatch the `setValue` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

Java

```
public class GridI implements Grid
{
    ...

    public synchronized int[][] getGrid(Current current)
    {
        return _grid;
    }

    public synchronized void
setValue(int x, int y, int val, Current current)
    {
        int[][] newGrid = // shallow copy...
        newGrid[x][y] = val;
        _grid = newGrid;
    }

    ...
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `marshaled-result` metadata directive changes the signature of the corresponding servant method, but only if that operation returns one or more of the mutable types listed earlier. The metadata directive has the following effects:

- For an operation `op` that returns multiple values and at least one of those values has a mutable type, the name of the generated `OpResult` class becomes `OpMarshaledResult` instead and the return type of the servant method becomes `OpMarshaledResult`.
- For an operation `op` that returns a single value whose type is mutable, the Slice compiler generates an `OpMarshaledResult` class and the return type of the servant method becomes `OpMarshaledResult`.
- The constructor for `OpMarshaledResult` takes an extra argument of type `Current`. The servant must supply the `Current` in order for the results to be marshaled correctly.

The metadata directive also affects the asynchronous mapping but has no effect on the proxy mapping, nor does it affect the servant mapping of Slice operations that return `void` or return only immutable values.

> You can also annotate an interface with the `marshaled-result` metadata and it will be applied to all of the interface's operations.

After applying the metadata, we can now implement the `Grid` servant as follows:

Java

```
public class GridI implements Grid
{
    ...

    public synchronized Grid.GetGridMarshaledResult getGrid(Current
current)
    {
        return new Grid.GetGridMarshaledResult(_grid, curr); // _grid is
marshaled immediately
    }

    public synchronized void setValue(int x, int y, int val, Current
current)
    {
        _grid[x][y] = val; // this is safe
    }

    ...
}
```

Here are more examples to demonstrate the mapping:

Slice

```
class C { ... }
struct S { ... }
sequence<string> Seq;

interface Example
{
    C getC();

    ["marshaled-result"]
    C getC2();

    void getS(out S val);

    ["marshaled-result"]
    void getS2(out S val);

    string getValues(string name, out Seq val);

    ["marshaled-result"]
    string getValues2(string name, out Seq val);

    ["amd", "marshaled-result"]
    string getValuesAMD(string name, out Seq val);
}
```

Review the generated code below to see the changes that the presence of the metadata causes in the servant method signatures:

Java

```
public interface Example extends com.zeroc.Ice.Object
{
    public static class GetC2MarshaledResult
    {
        public GetC2MarshaledResult(C returnValue, Current current);
        ...
    }

    public static class GetS2MarshaledResult
    {
        public GetS2MarshaledResult(S returnValue, Current current);
        ...
    }

    public static class GetValuesResult
    {
        public GetValuesResult();
        public GetValuesResult(String returnValue, String[] val);
```

```
        public String returnValue;
        public String[] val;
    }

    public static class GetValues2MarshaledResult
    {
        public GetValues2MarshaledResult(String returnValue, String[]
val, Current current);
        ...
    }

    public static class GetValuesAMDResult
    {
        public GetValuesAMDResult()
        {
        }

        public GetValuesAMDResult(String returnValue, String[] val)
        {
            this.returnValue = returnValue;
            this.val = val;
        }

        public String returnValue;
        public String[] val;
    }

    public static class GetValuesAMDMarshaledResult implements
com.zeroc.Ice.MarshaledResult
    {
        public GetValuesAMDMarshaledResult(String returnValue, String[]
val, com.zeroc.Ice.Current current);
        ...
    }

    C getC(com.zeroc.Ice.Current current);
    GetC2MarshaledResult getC2(com.zeroc.Ice.Current current);
    S getS(com.zeroc.Ice.Current current);
    GetS2MarshaledResult getS2(com.zeroc.Ice.Current current);
    GetValuesResult getValues(String name, com.zeroc.Ice.Current
current);
    GetValues2MarshaledResult getValues2(String name,
com.zeroc.Ice.Current current);
```

```
    java.util.concurrent.CompletionStage<GetValuesAMDMarshaledResult>
getValuesAMDAsync(String name, com.zeroc.Ice.Current current);
}
```

See Also

- Server-Side Java Mapping for Interfaces
- Java Mapping for Operations
- Raising Exceptions in Java
- The Current Object

Raising Exceptions in Java

Servant Exceptions in Java

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

Java

```
@Override
public void write(String[] text, Current current)
    throws GenericError
{
    try
    {
        // Try to write file contents here...
    }
    catch(Exception ex)
    {
        throw new GenericError("Exception during write operation", ex);
    }
}
```

Note that, for this example, we have supplied the optional second parameter to the `GenericError` constructor. This parameter sets the inner exception and preserves the original cause of the error for later diagnosis.

If you throw an arbitrary Java run-time exception (such as a `ClassCastException`), the Ice run time catches the exception and then returns an `UnknownException` to the client.

The server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw an Ice run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

JVM Error Semantics

Servant implementations might inadvertently raise low-level errors during the course of their operation. These exceptions, which are subclasses of `java.lang.Error`, should not normally be trapped by the servant because they often indicate the occurrence of a serious, unrecoverable situation. For example, the JVM throws `StackOverflowError` when a recursive method has used all available stack space.

The Ice run time traps instances of `java.lang.Error` thrown by servants and then attempts to log the exception and send an `UnknownException` to the client. This attempt may or may not succeed, depending on the nature of the error and the condition of the JVM. As an example, the servant implementation might raise `OutOfMemoryError` and Ice's attempt to log the error and send a response could also fail due to lack of memory.

For an occurrence of `OutOfMemoryError` or `AssertionError`, Ice does not re-throw the error after logging a message and sending a response. For all other subclasses of `Error`, Ice re-throws the error so that the JVM's normal error-handling strategy will execute.

> When the JVM raises a subclass of `Error`, it usually means that a significant problem has occurred. Ice tries to send an `UnknownException` to the client (for twoway requests) in order to prevent the client from waiting indefinitely for a response. However, the

JVM may prevent Ice from successfully sending this response, which is another reason your clients should use invocation timeouts as a defensive strategy. Finally, in nearly all occurrences of an error, the server is unlikely to continue operating correctly even if Ice is able to complete the client's request. For example, the JVM terminates the thread that raised an uncaught error. In the case of an uncaught error raised by a servant, the thread being terminated is normally from the Ice server-side thread pool. If all of the threads in this pool eventually terminate due to uncaught errors, the server can no longer respond to new client requests.

See Also

- Run-Time Exceptions
- Java Mapping for Exceptions
- Server-Side Java Mapping for Interfaces
- Parameter Passing in Java

Object Incarnation in Java

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a Java Servant

Instantiating a servant means to allocate an instance:

Java

```
Node servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance and assigns its address to a reference of type `Node`. This works because `NodeI` is derived from `Node`, so a `Node` reference can refer to an instance of type `NodeI`. However, if we want to invoke a member function of the `NodeI` class at this point, we must use a `NodeI` reference:

Java

```
NodeI servant = new NodeI("Fred");
```

Whether you use a `Node` or a `NodeI` reference depends purely on whether you want to invoke a member function of the `NodeI` class: if not, a `Node` reference works just as well as a `NodeI` reference.

Creating an Identity in Java

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

Java

```
Identity id = new Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

Activating a Java Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

Java

```
_adapter.add(servant, id);
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

UUIDs as Identities in Java

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. Java provides a helper function that we can use to create such identities:

Java

```
public class Example
{
    public static void main(String[] args)
    {
        System.out.println(java.util.UUID.randomUUID().toString());
    }
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `randomUUID` creates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

Java

```
_adapter.addWithUUID(new NodeI("Fred"));
```

Creating Proxies in Java

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Hello World Application. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in Java

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write the following:

Java

```
NodePrx proxy = NodePrx.uncheckedCast(_adapter.addWithUUID(new NodeI("F
red")));
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `ObjectPrx`.

Direct Proxy Creation in Java

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

Java

```
Identity id = new Identity();
id.name = java.util.UUID.randomUUID().toString();
ObjectPrx o = _adapter.createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `randomUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Hello World Application
- Server-Side Java Mapping for Interfaces
- Object Adapter States
- Servant Locators
- Object Life Cycle

Asynchronous Method Dispatch (AMD) in Java

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's t hread pool. If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server can provide its results to the Ice run time for delivery to the client.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

On this page:

Enabling AMD with Metadata in Java

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *repl aces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

Slice

```
["amd"] interface I
{
    bool isValid();
    float computeRate();
}

interface J
{
    ["amd"] void startProcess();
    int endProcess();
}
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface) minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

AMD Mapping in Java

The mapping for an AMD operation is very similar to the synchronous mapping for an operation. There are two differences: the method's name has an `Async` suffix, and it returns `java.util.concurrent.CompletionStage<T>` where `T` is the actual return type as defined by the parameter mapping. The implementation of the operation, which typically returns an instance of the derived class `java.util.concu`

rrent.CompletableFuture<T>, must eventually complete the future by supplying either the results or an exception.

Here's a simple example to show the mapping for several operations:

Slice

```
["amd"]
interface I
{
    void opVoid();
    string opStringRet();
    void opStringIn(string s);
    void opStringOut(out string s);
    string opStringAll(string s, out string os);
}
```

Since we annotated the interface with the amd metadata, all of the operations use the asynchronous mapping:

Java

```
public interface I extends com.zeroc.Ice.Object
{
    public static class OpStringAllResult
    {
        public OpStringAllResult() {}
        public OpStringAllResult(String returnValue, String os)
        {
            this.returnValue = returnValue;
            this.os = os;
        }

        public String returnValue;
        public String os;
    }

    java.util.concurrent.CompletionStage<Void>
opVoidAsync(com.zeroc.Ice.Current current);
    java.util.concurrent.CompletionStage<java.lang.String>
opStringRetAsync(com.zeroc.Ice.Current current);
    java.util.concurrent.CompletionStage<Void> opStringInAsync(String s,
com.zeroc.Ice.Current current);
    java.util.concurrent.CompletionStage<java.lang.String>
opStringOutAsync(com.zeroc.Ice.Current current);
    java.util.concurrent.CompletionStage<I.OpStringAllResult>
opStringAllAsync(String s, com.zeroc.Ice.Current current);
    ...
}
```

AMD Thread Safety in Java

As with the synchronous mapping, you can add the `marshaled-result` metadata to operations that return mutable types in order to avoid potential thread-safety issues. The return type of your operation will change to be `CompletionStage<`*Op*`MarshaledResult>`.

AMD Exceptions in Java

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).

> These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

Although we recommend that the future be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the future or callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

AMD Example in Java

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

Slice

```
module Demo
{
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {}

    interface Model
    {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    }
}
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class implements `Demo.Model` and supplies a definition for the `interpolateAsync` method that creates a `Job` to hold the callback object and arguments, and adds the `Job` to a queue. The method is synchronized to guard access to the queue:

Java

```
public final class ModelI implements Demo.Model
{
    synchronized public java.util.concurrent.CompletionStage<float[][]>
    interpolateAsync(float[][] data, float factor,
com.zeroc.Ice.Current current)
        throws Demo.RangeError
    {
        java.util.concurrent.CompletableFuture<float[][]> future = new
java.util.concurrent.CompletableFuture<float[][]>();
        _jobs.add(new Job(future, data, factor));
        return future;
    }

    java.util.LinkedList<Job> _jobs = new java.util.LinkedList<Job>();
}
```

After queuing the information, the implementation returns an uncompleted future to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

Java

```
class Job
{
    Job(java.util.concurrent.CompletableFuture<float[][]> future,
float[][] grid, float factor)
    {
        _future = future;
        _grid = grid;
        _factor = factor;
    }

    void execute()
    {
        if(!interpolateGrid())
        {
            _future.completeExceptionally(new Demo.RangeError());
        }
        else
        {
            _future.complete(_grid);
        }
    }

    private boolean interpolateGrid()
    {
        // ...
    }

    private java.util.concurrent.CompletableFuture<float[][]> _future;
    private float[][] _grid;
    private float _factor;
}
```

If `interpolateGrid` returns `false`, then we complete the future to indicate that a range error has occurred. If interpolation was successful, we send the modified grid back to the client by calling `complete` on the future.

See Also

- User Exceptions
- Run-Time Exceptions
- Asynchronous Method Invocation (AMI) in Java
- The Ice Threading Model

Example of a File System Server in Java

This page presents the source code for a Java server that implements our file system and communicates with the client we wrote earlier. The code is fully functional, apart from the required interlocking for threads.

Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

On this page:

Implementing a File System Server in Java

We have now seen enough of the server-side Java mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of three source files:

- `Server.java`
  This file contains the server main program.
- `Filesystem/DirectoryI.java`
  This file contains the implementation for the `Directory` servants.
- `Filesystem/FileI.java`
  This file contains the implementation for the `File` servants.

Server Main Program in Java

Our server main program, in the file `Server.java`, consists of two static methods, `main` and `run`. `main` creates and destroys an Ice communicator, and `run` uses this communicator instantiate our file system objects:

Java

```
import Filesystem.*;

public class Server
{
    public static void main(String[] args)
    {
        int status = 0;
        java.util.List<String> extraArgs = new
java.util.ArrayList<String>();

        //
        // Try with resources block - communicator is automatically
destroyed
        // at the end of this try block
        //
        try(com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(args, extraArgs))
```

```
        {
            //
            // Install shutdown hook to (also) destroy communicator
during JVM shutdown.
            // This ensures the communicator gets destroyed when the
user interrupts the application
            // with Ctrl-C.
            //
            Runtime.getRuntime().addShutdownHook(new Thread(() ->
            {
                communicator.destroy();
                System.err.println("terminating");
            }));

            if(!extraArgs.isEmpty())
            {
                System.err.println("too many arguments");
                status = 1;
            }
            else
            {
                status = run(communicator);
            }
        }

        System.exit(status);
    }

    private static int run(com.zeroc.Ice.Communicator communicator)
    {
        //
        // Create an object adapter.
        //
        com.zeroc.Ice.ObjectAdapter adapter =

communicator.createObjectAdapterWithEndpoints("SimpleFilesystem",
"default -h localhost -p 10000");

        //
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI root = new DirectoryI(communicator, "/", null);
        root.activate(adapter);

        //
        // Create a file called "README" in the root directory
        //
        FileI file = new FileI(communicator, "README", root);
        String[] text;
        text = new String[]{ "This file system contains a collection of
```

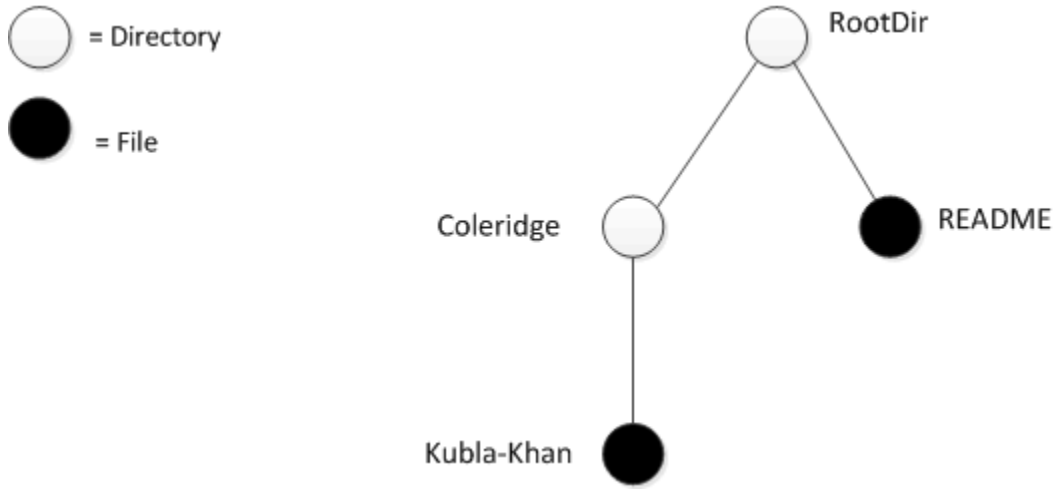
```
poetry." };
        try
        {
            file.write(text, null);
        }
        catch(GenericError e)
        {
            System.err.println(e.reason);
        }
        file.activate(adapter);

        //
        // Create a directory called "Coleridge" in the root directory
        //
        DirectoryI coleridge = new DirectoryI(communicator, "Coleridge",
root);
        coleridge.activate(adapter);

        //
        // Create a file called "Kubla_Khan" in the Coleridge directory
        //
        file = new FileI(communicator, "Kubla_Khan", coleridge);
        text = new String[]{ "In Xanadu did Kubla Khan",
                             "A stately pleasure-dome decree:",
                             "Where Alph, the sacred river, ran",
                             "Through caverns measureless to man",
                             "Down to a sunless sea." };
        try
        {
            file.write(text, null);
        }
        catch(GenericError e)
        {
            System.err.println(e.reason);
        }
        file.activate(adapter);

        //
        // All objects are created, allow client requests now
        //
        adapter.activate();

        //
        // Wait until we are done
        //
        communicator.waitForShutdown();

        return 0;
    }
```

```
}
```

The code imports the contents of the `Filesystem` package. This avoids having to continuously use fully-qualified identifiers with a `Filesystem.` prefix.

The next part of the source code is the definition of the `Server` class. Much of this code is boiler plate: we create a communicator, then an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

Java

```
DirectoryI root = new DirectoryI("/", null);
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in the above illustration:

Java

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI("/", null);

// Create a file "README" in the root directory
//
File file = new FileI("README", root);
String[] text;
text = new String[]
{
    "This file system contains a collection of poetry."
};
try
{
    file.write(text, null);
}
catch(GenericError e)
{
    System.err.println(e.reason);
}

// Create a directory "Coleridge" in the root directory
//
DirectoryI coleridge = new DirectoryI("Coleridge", root);

// Create a file "Kubla_Khan" in the Coleridge directory
//
file = new FileI("Kubla_Khan", coleridge);
text = new String[]{ "In Xanadu did Kubla Khan",
                     "A stately pleasure-dome decree:",
                     "Where Alph, the sacred river, ran",
                     "Through caverns measureless to man",
                     "Down to a sunless sea." };
try
{
    file.write(text, null);
}
catch(GenericError e)
{
    System.err.println(e.reason);
}
```

We first create the root directory and a file `README` within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

Java

```
        String[] text;
        text = new String[]
        {
            "This file system contains a collection of poetry."
        };
        try
        {
            file.write(text, null);
        }
        catch(GenericError e)
        {
            System.err.println(e.reason);
        }
```

Recall that Slice sequences by default map to Java arrays. The Slice type `Lines` is simply an array of strings; we add a line of text to our `README` file by initializing the `text` array to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by writing:

Java

```
            file.write(text, null);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Java function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the illustration listed above.

`FileI` Servant Class in Java

Our `FileI` servant class has the following basic structure:

Java

```
public class FileI implements File
{
    // Constructor and operations here...

    public static com.zeroc.Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private String[] _lines;
}
```

The class has a number of data members:

- `_adapter`
  This static member stores a reference to the single object adapter we use in our server.
- `_name`
  This member stores the name of the file incarnated by the servant.
- `_parent`
  This member stores the reference to the servant for the file's parent directory.
- `_lines`
  This member holds the contents of the file.

The `_name` and `_parent` data members are initialized by the constructor:

Java

```
public FileI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    assert(_parent != null);

    // Create an identity
    //
    com.zeroc.Ice.Identity myID = new com.zeroc.Ice.Identity();
    myID.name = java.util.UUID.randomUUID().toString();

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and
    // add it as a child to the parent
    //
    NodePrx thisNode
= NodePrx.uncheckedCast(_adapter.createProxy(myID));
    _parent.addChild(thisNode);
}
```

After initializing the `_name` and `_parent` members, the code verifies that the reference to the parent is not null because every file must have a parent directory. The constructor then generates an identity for the file by calling `java.util.UUID.randomUUID` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function in `DirectoryI` Methods.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

Java

```
// Slice Node::name() operation

public String name(com.zeroc.Ice.Current current)
{
    return _name;
}

// Slice File::read() operation

public String[] read(com.zeroc.Ice.Current current)
{
    return _lines;
}

// Slice File::write() operation

public void write(String[] text, com.zeroc.Ice.Current current)
    throws GenericError
{
    _lines = text;
}
```

The `name` method is inherited from the generated `Node` interface. It returns the value of the `_name` member.

The `read` and `write` methods are inherited from the generated `File` interface and return and set the `_lines` member.

`DirectoryI` Servant Class in Java

The `DirectoryI` class has the following basic structure:

Java

```
package Filesystem;

public final class DirectoryI implements Directory
{
    // Constructor and operations here...

    public static com.zeroc.Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private java.util.ArrayList<NodePrx> _contents
= new java.util.ArrayList<NodePrx>();
}
```

`DirectoryI` Data Members

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds a null reference.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

Java

```
public DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    // Create an identity. The parent has the fixed identity "RootDir"
    //
    com.zeroc.Ice.Identity myID = new com.zeroc.Ice.Identity();
    myID.name = _parent != null ? java.util.UUID.randomUUID().toString(
) : "RootDir";

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and add it as a
child to the parent
    //
    NodePrx thisNode
= NodePrx.uncheckedCast(_adapter.createProxy(myID));
    if(_parent != null)
    {
        _parent.addChild(thisNode);
    }
}
```

`DirectoryI` Constructor

The constructor creates an identity for the new directory by calling `java.util.UUID.randomUUID`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a reference to itself and passes it to the `addChild` helper function.

`DirectoryI` Methods

`addChild` adds the passed reference to the `_contents` list:

Java

```
void addChild(NodePrx child)
{
    _contents.add(child);
}
```

The remainder of the operations, `name` and `list`, are equally trivial:

Java

```
public String name(com.zeroc.Ice.Current current)
{
    return _name;
}

// Slice Directory::list() operation

public NodePrx[] list(com.zeroc.Ice.Current current)
{
    NodePrx[] result = new NodePrx[_contents.size()];
    _contents.toArray(result);
    return result;
}
```

Note that the `_contents` member is of type `java.util.ArrayList<NodePrx>`, which is convenient for the implementation of the `addChild` method. However, this requires us to convert the list into a Java array in order to return it from the `list` operation.

See Also

- Slice for a Simple File System
- Example of a File System Client in Java
- Java Mapping for Sequences
- The Ice Threading Model

Slice-to-Java Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to Java is identical to the mapping for these constructs without the `local` qualifier. The generated Java code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

The rest of this section describes the mapping of the remaining local types to Java:

- Java Mapping for Local Interfaces
- Java Mapping for Local Classes
- Java Mapping for Local Exceptions
- Java Mapping for Operations on Local Types
- Java Mapping for Data Members in Local Types

Java Mapping for Local Interfaces

On this page:

Mapped Java Class

A Slice local interface is mapped to a Java interface with the same name, for example:

Slice

```
[["java:package:com.zeroc"]]

module Ice
{
    local interface Communicator
    {
        ...
    }
}
```

is mapped to the Java interface `Communicator`:

Java

```
package com.zeroc.Ice;

public interface Communicator
{
    ...
}
```

The `delegate` metadata allows you to map a local interface with a single operation to a functional Java interface. For example:

Slice

```
[["java:package:com.zeroc"]]
module Ice
{
    ["delegate"] local interface ValueFactory
    {
        Value create(string type);
    }
}
```

is mapped to:

Java

```
package com.zeroc.Ice;

@FunctionalInterface
public interface ValueFactory
{
    com.zeroc.Ice.Value create(String type);
}
```

LocalObject in Java

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to `java.lang.Object` in Java.

Mapping for Local Interface Inheritance in Java

Inheritance of local Slice interfaces is mapped to interface inheritance in Java. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

Java

```
package M;

public interface A {}
public interface B extends A {}
public interface C extends A {}
public interface D extends B, C {}
```

Java Mapping for Local Classes

On this page:

Mapped Java Class

A local Slice class is mapped to a concrete or abstract Java class with the same name. For example:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ...
    }
}
```

is mapped to the Java class `ConnectionInfo`:

Java

```
package com.zeroc.Ice;

public class ConnectionInfo implements java.lang.Cloneable
{
    ...
}
```

All mapped Java classes implement `java.lang.Cloneable.`

LocalObject in Java

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to `java.lang.Object` in Java.

Mapping for Local Interface Inheritance in Java

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` for classes is mapped to `extends` in Java, and `implements` is mapped to `implements`. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

Java

```
package M;

public interface A {}
public interface B {}

public class C implements A, B, java.lang.Cloneable {}
public class D extends C {}
```

Java Mapping for Local Exceptions

On this page:

Mapped Java Class

A local Slice exception is mapped to a Java class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the Java class `InitializationException`:

Java

```
package com.zeroc.Ice;

public class InitializationException extends LocalException
{
    ...
}
```

Base Class for Local Exceptions in Java

All mapped Java classes for local exceptions extend the abstract class `LocalException`:

Java

```
package com.zeroc.Ice;

public abstract class LocalException extends Exception
{
    public LocalException() {}
    public LocalException(Throwable cause) { ... }
}
```

Mapping for Local Exception Inheritance in Java

A local Slice exception can extend another Slice exception, which is mapped to `extends` in Java. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

Java

```
package M;
public class ErrorBase extends com.zeroc.Ice.LocalException { ... }
public class ResourceError extends ErrorBase { ... }
```

Java Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to a Java method with the same name. The mapping of operation parameters to Java is identical to the Client-Side Mapping for these parameters, in particular, when an operation has more than one return value and out parameters, the mapped Java method returns a generated `Result` object.

Unlike the Client-Side mapping, there is no mapped method with a trailing Context parameter.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

Java

```
package M;

public interface L
{
    public static class OpResult
    {
        public OpResult()
        {
        }

        public OpResult(String returnValue, int m, String t, L[]
newLSeq)
        {
            this.returnValue = returnValue;
            this.m = m;
            this.t = t;
            this.newLSeq = newLSeq;
        }

        public String returnValue;
        public int m;
        public String t;
        public L[] newLSeq;
    }

    OpResult op(int n, String s, java.lang.Object any);
}
```

Java Mapping for Data Members in Local Types

Data members on local Slice types (classes, exceptions and structs) are mapped to Java just like the data members of the corresponding non local Slice construct.

A local Slice type can have a data member of type local interface or class: it is mapped a Java data member with the mapped Java interface or class as its type.

The Util Class in Java

Ice for Java includes a number of utility APIs in the `com.zeroc.Ice.Util` class. This page summarizes the contents of these APIs for your reference.

The `Util` Class

Communicator Initialization Methods

`Util` provides a number of overloaded `initialize` methods that create a communicator.

Identity Conversion

`Util` contains two methods for converting object identities of type `Identity` to and from strings.

Per-Process Logger Methods

`Util` provides methods for getting and setting the per-process logger.

Property Creation Methods

`Util` provides a number of overloaded `createProperties` methods that create property sets.

Proxy Comparison Methods

Two methods, `proxyIdentityCompare` and `proxyIdentityAndFacetCompare`, allow you to compare object identities that are stored in proxies (either ignoring the facet or taking the facet into account).

Version Information

The `stringVersion` and `intVersion` methods return the version of the Ice run time:

Java

```
public static String stringVersion();
public static int intVersion();
```

The `stringVersion` method returns the Ice version in the form *<major>.<minor>.<patch>*, for example, `3.7.1`. For beta releases, the version is *<major>.<minor>*b, for example, `3.7b`.

The `intVersion` method returns the Ice version in the form *AABBCC*, where *AA* is the major version number, *BB* is the minor version number, and *CC* is patch level, for example, `30701` for version 3.7.1. For beta releases, the patch level is set to 51 so, for example, for version 3.7b, the value is `30751`.

See Also

- Command-Line Parsing and Initialization
- Setting Properties
- Object Identity

Custom Class Loaders

Certain features of the Ice for Java run-time necessitate dynamic class loading. Applications with special requirements can supply a custom class loader for Ice to use in the following situations:

- Unmarshaling user exceptions and instances of concrete Slice classes
- Loading Ice plug-ins
- Loading IceSSL certificate verifiers and password callbacks

If an application does not supply a class loader (or if the application-supplied class loader fails to locate a class), the Ice run time attempts to load the class using class loaders in the following order:

- current thread's class loader
- default class loader (that is, by calling `Class.forName`)
- system class loader

Note that an application must install value factories for any abstract Slice classes it might receive, regardless of whether the application also installs a custom class loader.

To install a custom class loader, set the `classLoader` member of `com.zeroc.Ice.InitializationData` prior to creating a communicator:

Java

```
com.zeroc.Ice.InitializationData initData = new com.zeroc.Ice.Initializ
ationData();
initData.classLoader = new MyClassLoader();
com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(args, initData);
```

See Also

- Java Mapping for Exceptions
- Java Mapping for Classes
- Plug-in Facility
- IceSSL
- Communicator Initialization

Java Interrupts

Java applications can safely interrupt a thread that calls a blocking Ice API.

> You must enable the `Ice.ThreadInterruptSafe` property to use interrupts. Enabling interrupts causes Ice to disable message buffer caching, which may incur a slight performance penalty.

With interrupt support enabled, Ice guarantees that every call into the run time that could potentially block indefinitely is an *interruption point*. On entry, an interruption point raises the unchecked exception `OperationInterruptedException` immediately if the current thread's interrupted flag is true. If the application interrupts the run time after the interruption point has begun its processing, the interruption point will either raise `OperationInterruptedException` or return control to the application with the thread's interrupted flag set. Under no circumstances will Ice silently discard an interrupt.

For example, the following code shows how to interrupt a thread that's blocked while making a synchronous proxy invocation:

Java

```
Thread proxyThread = Thread.currentThread();
AccountPrx account = // get proxy...

try
{
    String name = account.getName(); // Blocks calling thread
    ...
}
catch(com.zeroc.Ice.OperationInterruptedException ex)
{
   // The invocation was interrupted.
}

// At this point either the invocation was interrupted with an
// OperationInterruptedException or proxyThread.isInterrupted() is true.

// In another thread either just prior to or during the getName()
invocation.
proxyThread.interrupt();
```

Here is a similar example that uses the asynchronous proxy API instead:

Java

```
Thread proxyThread = Thread.currentThread();
AccountPrx account = // get proxy...

CompletableFuture<String> f = account.getNameAsync(); // Never blocks

try
{
    // do more work
    String name = f.join(); // May block calling thread
    ...
}
catch(java.util.concurrent.CompletionException ex)
{
    if(ex.getCause() instanceof
com.zeroc.Ice.OperationInterruptedException)
    {
        // The invocation was interrupted.
    }
}

// At this point either the invocation was interrupted with an
// OperationInterruptedException or proxyThread.isInterrupted() is true.

// In another thread either just prior to or during the getName()
invocation.
proxyThread.interrupt();
```

Synchronous proxy invocations are interruption points. For an asynchronous proxy invocation, the `Async` method never blocks and therefore it is not an interruption point.

Additionally, all of the methods listed in Blocking API Calls are also interruption points. In the specific case of `Communicator.destroy()`, we recommend calling `destroy` in a loop as shown below:

Java

```
while(true)
{
    try
    {
        communicator.destroy();
        break;
    }
    catch(com.zeroc.Ice.OperationInterruptedException ex)
    {
        continue;
    }
}
```

Interrupting a call to `destroy` could leave the Ice run time in a partially-destroyed state; calling `destroy` again as we've done here ensures Ice can finish reclaiming the communicator's resources.

See Also

- Blocking API Calls
- Java Mapping

Java Compat Mapping

Topics

Selecting the Java Compat Mapping

Ice provides two distinct Java mappings:

- Java Compat
  This mapping is largely backward-compatible with prior Ice releases. Although Ice 3.7 no longer supports Java versions prior to Java 8, the "Compat" mapping does not depend on any Java 8-specific language or run-time features.

- Java
  This is a new mapping that takes advantage of features in Java 8.

This chapter describes the Java Compat mapping.

> The Java Compat mapping is provided for backwards compatibility, and will not be included in future releases. If you are starting a new project with Ice, please use the Java mapping; the only exception is if you need to support older Android devices, in which case you'll need to use the Java Compat mapping. If you are upgrading an existing application, you should consider upgrading your code to use the Java mapping.

Selecting the Java Compat Mapping

slice2java, the Slice-to-Java translator, generates code for the Java mapping by default. You must specify the `--compat` option in order to generate code for the Java Compat mapping instead.

Initialization in Java Compat

Every Ice-based application needs to initialize the Ice run time, and this initialization returns a `Ice.Communicator` object.

A `Communicator` is a local Java object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application.

You initialize the Ice run time by calling `Ice.Util.initialize`, for example:

Java

```
public static void main(String[] args)
{
    Ice.Communicator communicator = Ice.Util.initialize(args);
    ...
}
```

`Util.initialize` accepts the argument vector that is passed to `main` by the operating system. The method scans the argument vector for any command-line options that are relevant to the Ice run time. If anything goes wrong during initialization, `Util.initialize` throws an exception.

> The semantics of Java arrays prevents this simple `Util.initialize` from modifying the argument vector. You can use another overload of `Util.initialize` to receive an argument vector with all Ice-related arguments removed.

Once you no longer need a `Communicator`, you must call `destroy` on this `Communicator`. The `destroy` method is responsible for finalizing the instance of the Ice run time embodied by this Communicator. In particular, in an Ice server, `destroy` waits for operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your application to terminate without calling `destroy` first.

The general shape of our `main` method becomes:

Java

```
public class App
{
    public static void main(String[] args)
    {
        int status = 0;
        Ice.Communicator communicator = Ice.Util.initialize(args);

        //
        // Register shutdown hook to destroy communicator during JVM
shutdown
        //
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
communicator.destroy(); }));

        // ...
        System.exit(status);
    }
}
```

The only pitfall with the code above is if you neglect to call `System.exit`, the application will continue to run because the Communicator started non-daemon threads.

Another way to initialize and destroy a Communicator is with a try-with-resources statement:

Java

```
public class App
{
    public static void main(String[] args)
    {
        int status = 0;
        try(Ice.Communicator communicator = Ice.Util.initialize(args))
        {
            // ...
        } // communicator is destroyed automatically here
        System.exit(status);
    }
}
```

> The `Communicator` interface implements `java.lang.AutoCloseable`: at the end of a try-with-resources statement, the communicator is closed (destroyed) automatically, without an explicit call to the `destroy` method.

If you initialize your communicator in a try-with-resources statement, you may also add a shutdown hook to destroy the communicator. `destroy` does not throw any exception and calling `destroy` multiple times is perfectly ok. A shutdown hook that calls `destroy` is useful to interrupt long-running Ice invocations when the application receives a user interrupt such as Ctrl-C.

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-Java Compat Mapping

In this section, we present the client-side Slice-to-Java Compat mapping. The mapping defines how Slice data types are translated to Java types, and how clients invoke operations, pass parameters, and handle errors. Much of the Java Compat mapping is intuitive. For example, Slice sequences map to Java arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in Java.

The Java API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the Java mapping, you should need no more than the Slice definition of your application and knowledge of the Java mapping rules. In particular, looking through the generated code in order to discern how to use the Java mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **Ice Packaging**
> All of the APIs for the Ice run time are nested in the `Ice` package to avoid clashes with definitions for other libraries or applications. Some of the contents of the package are generated from Slice definitions; other parts of the package provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of this package throughout the remainder of the manual. For the sake of brevity, the discussions and code examples usually omit the Ice package prefix.

Topics

Java Compat Mapping for Identifiers

A Slice identifier maps to an identical Java identifier. For example, the Slice identifier `Clock` becomes the Java identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a Java keyword or is an identifier reserved by the Ice run time (such as `checkedCast`), the corresponding Java identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.

> You should try to avoid such identifiers as much as possible.

A single Slice identifier often results in several Java identifiers. For example, for a Slice interface named `Foo`, the generated Java code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_while` and `whilePrx` (*not* `_whilePrx`), that is, the underscore prefix is applied only to those generated identifiers that actually require it.

See Also

- Lexical Rules

Java Compat Mapping for Modules

A Slice module maps to a Java package with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions. For example:

Slice

```
// Definitions at global scope here...

module M1
{
    // Definitions for M1 here...
    module M2
    {
        // Definitions for M2 here...
    }
}

// ...

module M1    // Reopen M1
{
    // More definitions for M1 here...
}
```

This definition maps to the corresponding Java definitions:

Java Compat

```
package M1;
// Definitions for M1 here...

package M1.M2;
// Definitions for M2 here...

package M1;
// Definitions for M1 here...
```

Note that these definitions appear in the appropriate source files; source files for definitions in module `M1` are generated in directory `M1` under neath the top-level directory, and source files for definitions for module `M2` are generated in directory `M1/M2` underneath the top-level directory. You can set the top-level output directory using the `--output-dir` option with `slice2java`.

See Also

- Modules
- Using the Slice Compilers

Java Compat Mapping for Built-In Types

The Slice built-in types are mapped to Java types as follows:

Slice	Java
bool	`boolean`
byte	`byte`
short	`short`
int	`int`
long	`long`
float	`float`
double	`double`
string	`String`

Mapping of Slice built-in types to Java.

See Also

- Basic Types

Java Compat Mapping for Enumerations

A Slice enumeration maps to the corresponding enumeration in Java. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The Java mapping for `Fruit` is shown below:

Java Compat

```
public enum Fruit implements java.io.Serializable
{
    Apple,
    Pear,
    Orange;

    public int value();

    public static Fruit valueOf(int v);

    // ...
}
```

Given the above definitions, we can use enumerated values as follows:

Java Compat

```
Fruit f1 = Fruit.Apple;
Fruit f2 = Fruit.Orange;

if(f1 == Fruit.Apple) // Compare with constant
{
    // ...
}

if(f1 == f2)          // Compare two enums
{
    // ...
}

switch(f2)            // Switch on enum
{
    case Fruit.Apple:
        // ...
        break;
    case Fruit.Pear:
        // ...
        break;
    case Fruit.Orange:
        // ...
        break;
}
```

The Java mapping includes two methods of interest. The `value` method returns the Slice value of an enumerator, which is not necessarily the same as its ordinal value. The `valueOf` method translates a Slice value into its corresponding enumerator, or returns `null` if no match is found. Note that the generated class contains a number of other members, which we have not shown. These members are internal to the Ice run time and you must not use them in your application code (because they may change from release to release).

In the `Fruit` definition above, the Slice value of each enumerator matches its ordinal value. This will not be true if we modify the definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The table below shows the new relationship between ordinal value and Slice value:

Enumerator	Ordinal	Slice
`Apple`	0	0
`Pear`	1	3
`Orange`	2	4

Java enumerated types inherit implicitly from `java.lang.Enum`, which defines methods such as `ordinal` and `compareTo` that operate on the *ordinal* value of an enumerator, not its Slice value.

See Also

- Enumerations

Java Compat Mapping for Structures

On this page:

Basic Java Mapping for Structures

A Slice structure maps to a Java class with the same name. For each Slice data member, the Java class contains a corresponding public data member. For example, here is our Employee structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The Slice-to-Java compiler generates the following definition for this structure:

Java Compat

```
public final class Employee implements java.lang.Cloneable,
java.io.Serializable
{
    public long number;
    public String firstName;
    public String lastName;

    public Employee() {}

    public Employee(long number, String firstName, String lastName)
    {
        this.number = number;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public boolean equals(java.lang.Object rhs)
    {
        // ...
    }

    public int hashCode()
    {
        // ...
    }

    public java.lang.Object clone()
    {
        java.lang.Object o;
        try
        {
            o = super.clone();
        }
        catch(java.lang.CloneNotSupportedException ex)
        {
            assert false; // impossible
        }
        return o;
    }
}
```

For each data member in the Slice definition, the Java class contains a corresponding public data member of the same name. Note that you can optionally customize the mapping for data members to use getters and setters instead.

The `equals` member function compares two structures for equality. Note that the generated class also provides the usual `hashCode` and `clone` methods. (`clone` has the default behavior of making a shallow copy.)

Java Default Constructors for Structures

Structures have a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default Java behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

Structures also have a second constructor that has one parameter for each data member. This allows you to construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members).

See Also

- Structures
- Customizing the Java Compat Mapping

Java Compat Mapping for Sequences

A Slice sequence maps to a Java array. This means that the Slice-to-Java compiler does not generate a separate named type for a Slice sequence.

For example:

Slice

```
sequence<Fruit> FruitPlatter;
```

This definition simply corresponds to the Java type `Fruit[]`. Naturally, because Slice sequences are mapped to Java arrays, you can take advantage of all the array functionality provided by Java, such as initialization, assignment, cloning, and the `length` member. For example:

Java Compat

```
Fruit[] platter = { Fruit.Apple, Fruit.Pear };
assert(platter.length == 2);
```

Alternate mappings for sequence types are also possible.

See Also

- Sequences
- Customizing the Java Compat Mapping

Java Compat Mapping for Dictionaries

Here is the definition of our EmployeeMap once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Java mapping does not create a separate named type for this definition. Instead, the dictionary is simply an instance of the generic type `java.util.Map<K, V>`, where *K* is the mapping of the key type and *V* is the mapping of the value type. In the example above, `EmployeeMap` is mapped to the Java type `java.util.Map<Long, Employee>`. The following code demonstrates how to allocate and use an instance of `EmployeeMap`:

Java Compat

```
java.util.Map<Long, Employee> em =
new java.util.HashMap<Long, Employee>();

Employee e = new Employee();
e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.put(e.number, e);
```

The type-safe nature of the mapping makes iterating over the dictionary quite convenient:

Java Compat

```
em.forEach((num, employee) ->
    {

System.out.println(employee.firstName + " was employee #" + num);
    });
```

Alternate mappings for dictionary types are also possible.

See Also

- Dictionaries
- Customizing the Java Compat Mapping

Java Compat Mapping for Constants

Here are the sample constant definitions once more:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

Java Compat

```
public interface AppendByDefault
{
    boolean value = true;
}

public interface LowerNibble
{
    byte value = 15;
}

public interface Advice
{
    String value = "Don't Panic!";
}

public interface TheAnswer
{
    short value = 42;
}

public interface PI
{
    double value = 3.1416;
}

public interface FavoriteFruit
{
    Fruit value = Fruit.Pear;
}
```

As you can see, each Slice constant is mapped to a Java interface with the same name as the constant. The interface contains a member named `value` that holds the value of the constant.

Slice string literals that contain non-ASCII characters or universal character names are mapped to Java string literals with universal character names. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

Java Compat

```
public interface Egg
{
    String value = "\u0153uf";
}

public interface Heart
{
    String value = "c\u0153ur";
}

public interface Banana
{
    String value = "\ud83c\udf4c";
}
```

See Also

- Constants and Literals

Java Compat Mapping for Exceptions

On this page:

Java Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

Java Compat

```
public class GenericError extends Ice.UserException
{
    public GenericError()
    {
        this.reason = "";
    }

    public GenericError(Throwable cause)
    {
        super(cause);
        this.reason = "";
    }

    public GenericError(String reason)
    {
        this.reason = reason;
    }

    public GenericError(String reason, Throwable cause)
    {
        super(cause);
        this.reason = reason;
    }

    public String
    ice_id()
```

```
    {
        return "::M::GenericError";
    }

    public String reason;

    ...
}

public class BadTimeVal extends GenericError
{
    public BadTimeVal()
    {
        super();
    }

    public BadTimeVal(Throwable cause)
    {
        super(cause);
    }

    public BadTimeVal(String reason)
    {
        super(reason);
    }

    public BadTimeVal(String reason, Throwable cause)
    {
        super(reason, cause);
    }

    public String
    ice_id()
    {
        return "::M::BadTimeVal";
    }

    ...
}

public class BadZoneName extends GenericError
{
    public BadZoneName()
    {
        super();
    }

    public BadZoneName(Throwable cause)
    {
        super(cause);
```

```
}

public BadZoneName(String reason)
{
    super(reason);
}

public BadZoneName(String reason, Throwable cause)
{
    super(reason, cause);
}

public String
ice_id()
{
    return "::M::BadZoneName";
}
```

```
    ...
}
```

Each Slice exception is mapped to a Java class with the same name. For each data member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.) A JavaBean-style API is used for optional data members, and you can customize the mapping to force required members to use this same API.

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception also defines the `ice_id` method, which returns the Slice type ID of the exception.

All user exceptions are derived from the base class `UserException`. This allows you to catch all user exceptions generically by installing a handler for `UserException`. `UserException`, in turn, derives from `java.lang.Exception`.

`UserException` implements a `clone` method that is inherited by its derived exceptions, so you can make member-wise shallow copies of exceptions.

Note that the generated exception classes contain other methods that are not shown. However, those methods are internal to the Java Compat mapping and are not meant to be called by application code.

Java Constructors for User Exceptions

Exceptions have a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default Java behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

If an exception declares or inherits any data members, the generated class provides a second constructor that accepts one parameter for each data member so that you can construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members). For a derived exception, this constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

The generated class may include an additional constructor if the exception declares or inherits any optional data members.

The Slice compiler generates overloaded versions of all constructors that accept a trailing `Throwable` argument for preserving an exception chain.

Java Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `LocalException` (which, in turn, derives indirectly from `java.lang.RuntimeException`).

`LocalException` implements a `clone` method that is inherited by its derived exceptions, so you can make member-wise shallow copies of exceptions.

Recall the inheritance diagram for user and run-time exceptions. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `UserException`
  This is the root of the inheritance tree for user exceptions.
- `TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `java.lang.Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- Java Mapping for Optional Data Members
- JavaBean Mapping
- Versioning
- Object Life Cycle

Java Compat Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Java Classes Generated for an Interface

The compiler generates quite a few source files for each Slice interface. In general, for an interface `<interface-name>`, the following source files are created by the compiler:

- `<interface-name>.java`
  This source file declares the `<interface-name>` Java interface.
- `<interface-name>Holder.java`
  This source file defines a holder type for the interface.
- `<interface-name>Prx.java`
  This source file defines the proxy interface `<interface-name>Prx`.
- `<interface-name>PrxHelper.java`
  This source file defines the helper type for the interface's proxy.
- `<interface-name>PrxHolder.java`
  This source file defines the holder type for the interface's proxy.
- `_<interface-name>Operations.java`
  `_<interface-name>OperationsNC.java`
  These source files each define an interface that contains the operations corresponding to the Slice interface. The `Operations` inter face is used for tie classes.

These are the files that contain code that is relevant to the client side. The compiler also generates a file that is specific to the server side, plus three additional files:

- `_<interface-name>Disp.java`
  This file contains the definition of the server-side skeleton class.
- `_<interface-name>Del.java`
- `_<interface-name>DelD.java`
- `_<interface-name>DelM.java`
  These files contain code that is internal to the Java mapping; they do not contain any functions of relevance to application programmers.

Proxy Interfaces in Java

On the client side, a Slice interface maps to a Java interface with methods that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Slice compiler generates the following definition for use by the client:

Java Compat

```
public interface SimplePrx extends ObjectPrx
{
    public void op();
    public void op(java.util.Map<String, String> context);
}
```

As you can see, the compiler generates a *proxy interface* `SimplePrx`. In general, the generated name is `<interface-name>Prx`. If an interface is nested in a module `M`, the generated class is part of package `M`, so the fully-qualified name is `M.<interface-name>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a proxy instance. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. For the preceding example, we find that the operation `op` has been mapped to the method `op`. Also note that `op` is overloaded: the second version of `op` has a parameter `context` of type `java.util.Map<String, String>`. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the `context` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Because all the `<interface-name>Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. The proxy references handed out by the Ice run time are always of type `<interface-name>Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Interface Inheritance in Java

Inheritance relationships among Slice interfaces are maintained in the generated Java classes. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

The generated code for `CPrx` reflects the inheritance hierarchy:

Java Compat

```
public interface CPrx extends APrx, BPrx
{
    ...
}
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

The `ObjectPrx` Interface in Java

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `ObjectPrx`. `ObjectPrx` provides a number of methods:

Java Compat

```
public interface ObjectPrx
{
    boolean equals(java.lang.Object r);
    Identity ice_getIdentity();
    boolean ice_isA(String id);
    boolean ice_isA(String id, java.util.Map<String, String> context);
    String[] ice_ids();
    String[] ice_ids(java.util.Map<String, String> context);
    String ice_id();
    String ice_id(java.util.Map<String, String> context);
    void ice_ping();
    void ice_ping(java.util.Map<String, String> context);
    // ...
}
```

The methods behave as follows:

- **`equals`**
  This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.
- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
}
```

To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

Java Compat

```
ObjectPrx o1 = ...;
ObjectPrx o2 = ...;
Identity i1 = o1.ice_getIdentity();
Identity i2 = o2.ice_getIdentity();

if(i1.equals(i2))
{
    // o1 and o2 denote the same object
}
else
{
    // o1 and o2 denote different objects

}
```

- **`ice_isA`**
  The `ice_isA` method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID. For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

Java Compat

```
ObjectPrx o = ...;
if(o != null && o.ice_isA("::Printer"))
{
    // o denotes a Printer object
}
else
{
    // o denotes some other type of object

}
```

Note that we are testing whether the proxy is null before attempting to invoke the `ice_isA` method. This avoids getting a `NullPoin`

terException if the proxy is null.

- **ice_ids**
  The ice_ids method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.
- **ice_id**
  The ice_id method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type BasePrx, with a static type ID of ::Base, the return value of ice_id might be ::Base, or it might something more derived, such as ::Derived.
- **ice_ping**
  The ice_ping method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

The ice_isA, ice_ids, ice_id, and ice_ping methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in ObjectPrx, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Proxy Helper Methods in Java

For each Slice interface, the Slice-to-Java compiler generates static helper methods that support down-casting and type discovery:

Java Compat

```
public static SimplePrx checkedCast(ObjectPrx b);
public static SimplePrx checkedCast(ObjectPrx b, java.util.Map<String,
String> context);
public static SimplePrx checkedCast(ObjectPrx b, String facet);

public static SimplePrx checkedCast(ObjectPrx b, String facet,
java.util.Map<String, String> context);
public static SimplePrx uncheckedCast(ObjectPrx b);
public static SimplePrx uncheckedCast(ObjectPrx b, String facet);
public static String ice_staticId();
```

For checkedCast, if the passed proxy is for an object of type Simple, or a proxy for an object with a type derived from Simple, the cast returns a non-null reference to a proxy of type SimplePrx; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference. Overloaded methods allow you to optionally specify a facet and a request context.

For the default Java mapping, these helper methods are generated in SimplePrx. For the Java Compat mapping, these helper methods are generated in SimplePrxHelper.

Given a proxy of any type, you can use a checkedCast to determine whether the corresponding object supports a given type, for example:

Java Compat

```
ObjectPrx obj = ...;        // Get a proxy from somewhere...

SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if(simple != null)
{
    // Object supports the Simple interface...
}
else
{
    // Object is not of type Simple...
}
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`. (This also explains the need for the helper method: the Ice run time must contact the server, so we cannot use a simple Java down-cast.)

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

Java Compat

```
ObjectPrx obj = ...;
// Get proxy...
ProcessPrx process
= ProcessPrxHelper.uncheckedCast(obj);  // No worries...
process.launch(40, 60);                                    // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a Java cast, the behavior is undefined.

Another method generated for every interface is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Using Proxy Methods in Java

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

Java Compat

```
ObjectPrx proxy = communicator.stringToProxy(...);
proxy = proxy.ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a `checkedCast` or `uncheckedCast` after using a factory method. However, the Java Compat mapping still requires a regular cast as shown in the example below:

Java Compat

```
ObjectPrx base = communicator.stringToProxy(...);
HelloPrx hello = HelloPrxHelper.checkedCast(base);
hello = (HelloPrx)hello.ice_invocationTimeout(10000); // Type is
preserved but a down-cast is required
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type using `checkedCast` or `uncheckedCast`.

Object Identity and Proxy Comparison in Java

Proxies provide an `equals` method that compares proxies:

Java Compat

```
interface ObjectPrx
{
    boolean equals(java.lang.Object r);
}
```

Note that proxy comparison with `equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `equals` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

Java Compat

```
ObjectPrx p1 = ...;        // Get a proxy...
ObjectPrx p2 = ...;        // Get another proxy...

if(!p1.equals(p2))
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Util` class:

Java Compat

```
public final class Util
{
    public static int proxyIdentityCompare(ObjectPrx lhs,
ObjectPrx rhs);
    public static int proxyIdentityAndFacetCompare(ObjectPrx lhs,
ObjectPrx rhs);
    // ...
}
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

Java Compat

```
ObjectPrx p1 = ...;        // Get a proxy...
ObjectPrx p2 = ...;        // Get another proxy...

if(Util.proxyIdentityCompare(p1, p2) != 0)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, `-1` if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

In addition, the Java mapping provides two wrapper classes that allow you to wrap a proxy for use as the key of a hashed collection:

Java Compat

```
public class ProxyIdentityKey
{
    public ProxyIdentityKey(ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public Ice.ObjectPrx getProxy();
}

public class ProxyIdentityFacetKey
{
    public ProxyIdentityFacetKey(ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public Ice.ObjectPrx getProxy();
}
```

The constructor caches the identity and the hash code of the passed proxy, so calls to `hashCode` and `equals` can be evaluated efficiently. The `getProxy` method returns the proxy that was passed to the constructor.

As for the comparison functions, `ProxyIdentityKey` only uses the proxy's identity, whereas `ProxyIdentityFacetKey` also includes the facet name.

Deserializing Proxies in Java

Proxy objects implement the `java.io.Serializable` interface that enables serialization of proxies to and from a byte stream. You can use the standard class `java.io.ObjectInputStream` to deserialize all Slice types *except* proxies; proxies are a special case because they must be created by a communicator.

To supply a communicator for use in deserializing proxies, an application must use the Ice-provided class `ObjectInputStream`:

Java Compat

```
public class ObjectInputStream extends java.io.ObjectInputStream
{
    public ObjectInputStream(Communicator communicator,
java.io.InputStream stream)
        throws java.io.IOException;

    public Communicator getCommunicator();
}
```

The code shown below demonstrates how to use this class:

Java Compat

```
Communicator communicator = ...
byte[] bytes = ... // data to be deserialized
java.io.ByteArrayInputStream byteStream =
new java.io.ByteArrayInputStream(bytes);
ObjectInputStream in = new ObjectInputStream(communicator, byteStream);
ObjectPrx proxy = (ObjectPrx)in.readObject();
```

Ice raises `java.io.IOException` if an application attempts to deserialize a proxy without supplying a communicator.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Type IDs
- Java Compat Mapping for Operations
- Request Contexts
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

Java Compat Mapping for Operations

On this page:

Basic Java Mapping for Operations

As we saw in the mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

Java Compat

```
NodePrx node = ...;             // Initialize proxy
String name = node.name();      // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

Normal and `idempotent` Operations in Java

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent  string op2();
}
```

The proxy interface for this is:

Java Compat

```
public interface ExamplePrx extends ObjectPrx
{
    String op1();
    String op2();
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

Passing Parameters in Java

In Parameters in Java

The parameter passing rules for the Java mapping are very simple: parameters are passed either by value (for simple types) or by reference (for complex types and type `string`). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy for these definitions:

Java Compat

```
public interface ClientToServerPrx extends ObjectPrx
{
    void op1(int i, float f, boolean b, String s);
    void op2(NumberAndString ns, String[] ss, java.util.Map<Long,
String[]> st);
    void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

Java Compat

```
ClientToServerPrx p = ...;              // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
boolean b = true;
String s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
String[] ss = { "Hello world!" };
java.util.Map<Long, String[]> st = new java.util.HashMap<Long,
String[]>();
st.put(0, ns);
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Out Parameters in Java

Java does not have pass-by-reference: parameters are always passed by value. For a function to modify one of its arguments, we must pass a reference (by value) to an object; the called function can then modify the object's contents via the passed reference.

To permit the called function to modify a parameter, the mapping uses *holder* classes. For example, for each of the built-in Slice types, such as `int` and `string`, the `Ice` package contains a corresponding holder class. Here are the definitions for the holder classes `Ice.IntHolder` and `Ice.StringHolder`:

Java Compat

```
package Ice;

public final class IntHolder
{
    public IntHolder() {}
    public IntHolder(int value)
    {
        this.value = value;
    }
    public int value;
}

public final class StringHolder
{
    public StringHolder() {}
    public StringHolder(String value)
    {
        this.value = value;
    }
    public String value;
}
```

A holder class has a public `value` member that stores the value of the parameter; the called function can modify the value by assigning to that member. The class also has a default constructor and a constructor that accepts an initial value.

For user-defined types, such as structures, the Slice-to-Java compiler generates a corresponding holder type. For example, here is the generated holder type for the `NumberAndString` structure we defined earlier:

Java Compat

```
public final class NumberAndStringHolder
{
    public NumberAndStringHolder() {}

    public NumberAndStringHolder(NumberAndString value)
    {
        this.value = value;
    }

    public NumberAndString value;
}
```

This looks exactly like the holder classes for the built-in types: we get a default constructor, a constructor that accepts an initial value, and the public `value` member.

Note that holder classes are generated for *every* Slice type you define. For example, for sequences, such as the `FruitPlatter` sequence, the compiler does not generate a special Java `FruitPlatter` type because sequences map to Java arrays. However, the compiler does generate a `FruitPlatterHolder` class, so we can pass a `FruitPlatter` array as an out-parameter.

To pass an out-parameter to an operation, we simply pass an instance of a holder class and examine the `value` member of each out-parameter when the call completes. Here are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Slice compiler generates the following code for these definitions:

Java Compat

```
public interface ServerToClientPrx extends Ice.ObjectPrx
{
    void op1(Ice.IntHolder i, Ice.FloatHolder f,
Ice.BooleanHolder b, Ice.StringHolder s);
    void op2(NumberAndStringHolder ns,
StringSeqHolder ss, StringTableHolder st);
    void op3(ClientToServerPrxHolder proxy);
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

Java Compat

```
ClientToServerPrx p = ...;              // Get proxy...

Ice.IntHolder ih = new Ice.IntHolder();
Ice.FloatHolder fh = new Ice.FloatHolder();
Ice.BooleanHolder bh = new Ice.BooleanHolder();
Ice.StringHolder sh = new Ice.StringHolder();
p.op1(ih, fh, bh, sh);

NumberAndStringHolder nsh = new NumberAndString();
StringSeqHolder ssh = new StringSeqHolder();
StringTableHolder sth = new StringTableHolder();
p.op2(nsh, ssh, sth);

ServerToClientPrxHolder stcph = new ServerToClientPrxHolder();
p.op3(stch);

System.out.writeln(ih.value);   // Show one of the values
```

Again, there are no surprises in this code: the various holder instances contain values once the operation invocation completes and the `value` member of each instance provides access to those values.

Null Parameters in Java

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid `NullPointerException`. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Optional Parameters in Java

The Java Compat mapping uses the generic class `Ice.Optional` to encapsulate optional parameters. All Slice types (including `string`) that map to Java reference types use the `Ice.Optional` generic class. To minimize garbage, the mapping also defines several non-generic classes to hold the unboxed primitive types `boolean`, `byte`, `short`, `int`, `long`, `float`, and `double`. For example, an optional `boolean` parameter maps to the `Ice.BooleanOptional` class.

Optional return values and output parameters are mapped to instances of `Ice.Optional` or one of the primitive wrapper classes, depending on their types. For operations with optional *input* parameters, the proxy provides a set of overloaded methods that accept them as optional values, and another set of methods that accept them as required values. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

The default Java mapping for this operation is shown below:

Java Compat

```
Ice.IntOptional execute(String params, Ice.FloatOptional value, ...);
Ice.IntOptional execute(Ice.Optional<String> params, Ice.FloatOptional
value, ...);
```

For cases where you are passing values for all optional input parameters, it is more efficient to use the required mapping and avoid creating temporary optional values.

A client can invoke `execute` as shown below:

Java Compat

```
Ice.IntOptional i;
Ice.FloatOptional v = new Ice.FloatOptional();

i = proxy.execute("--file log.txt", v);                          //
required mapping
i = proxy.execute(new Ice.Optional<String>("--file log.txt"), v); //
optional mapping
i = proxy.execute((Ice.Optional<String>)null, v);                //
params is unset

if(v.isSet())
{
    System.out.println("value = " + v.get());
}
```

Passing `null` where an `Ice.Optional` value is expected is equivalent to passing an `Ice.Optional` instance whose value is unset. For optional parameters of reference types, passing an explicit `null` requires a cast as shown in the last call to `execute` above.

Whereas the mapping for required output parameters uses `Holder` classes to transfer their values from the Ice run time to the caller, the optional mapping uses `Optional` values. When invoking a proxy operation, the client must pass an instance of the appropriate `Optional` type for each output parameter. The Ice run time invokes `clear` on the client's `Optional` value if the server does not supply a value for an output parameter, therefore it is safe to pass an `Optional` instance that already has a value.

A well-behaved program must not assume that an optional parameter always has a value. Calling `get` on an `Optional` instance for which no value is set raises `java.lang.IllegalStateException`.

Exception Handling in Java

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as Java exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

Java Compat

```
ChildPrx child = ...;   // Get child proxy...

try
{
    child.askToCleanUp();
}
catch(Tantrum t)
{
    System.out.write("The child says: ");
    System.out.writeln(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

Java Compat

```
public class Client
{
    public static void main(String[] args)
    {
        try
        {
            ChildPrx child = ...;   // Get child proxy...
            try
            {
                child.askToCleanUp();
                child.praise();     // Give positive feedback...
            }
            catch(Tantrum t)
            {
                System.out.print("The child says: ");
                System.out.println(t.reason);
                child.scold();      // Recover from error...
            }
        }
        catch(Ice.LocalException e)
        {
            e.printStackTrace();
        }
    }
}
```

Exceptions and Out-Parameters

For the Java Compat mapping, the Ice run time makes no guarantees about the state of out parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

> This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

See Also

- Operations
- Java Compat Mapping for Exceptions
- Java Compat Mapping for Interfaces
- Java Compat Mapping for Optional Data Members
- Collocated Invocation and Dispatch

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.

Java Compat Mapping for Classes

On this page:

Basic Java Mapping for Classes

A Slice class is mapped to a Java class with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions). Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

The Slice compiler generates the following code for this definition:

Java Compat

```
public class TimeOfDay extends Ice.ObjectImpl
{
    public short hour;
    public short minute;
    public short second;

    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second);

    // Server-side implementation of various Ice::Object operations:
    public boolean ice_isA(String s, Ice.Current __current);
    public String[] ice_ids(Ice.Current __current);
    public String ice_id(Ice.Current __current);

    public static String ice_staticId();

    public TimeOfDay clone();

    // ...
}
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits (indirectly) from `Ice.Object`. This means that all classes implicitly inherit from `Ice.Object`, which is the ultimate ancestor of all classes. Note that `Ice.Object` is *not* the same as `Ice.ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
   If a class has only data members, but no operations, the compiler generates a non-abstract class.
2. The generated class contains a public member for each Slice data member.
3. The generated class has a constructor that takes one argument for each data member, as well as a default constructor.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice::Object`

Classes implicitly inherit from a common base class, `Value`, which is mapped to `Ice.Object`. `Ice.Object` is the base class for all servants, as described in the Server-Side Java Compat Mapping for Interfaces.

`Ice.Object` contains a number of methods:

Java Compat

```
public interface Object
{
    boolean ice_isA(String s);
    boolean ice_isA(String s, Current current);

    void ice_ping();
    void ice_ping(Current current);

    String[] ice_ids();
    String[] ice_ids(Current current);

    String ice_id();
    String ice_id(Current current);

    void ice_preMarshal();
    void ice_postUnmarshal();
 SlicedData ice_getSlicedData();
    boolean ice_dispatch(Request request,
DispatchInterceptorAsyncCallback cb);
}
```

The `Ice.Object` methods behave as follows:

- `ice_isA`
  This method returns `true` if the object supports the given type ID, and `false` otherwise.
- `ice_ping`
  As for interfaces, `ice_ping` provides a basic reachability test for the class.
- `ice_ids`
  This method returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This method returns the actual run-time type ID for a class instance. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_preMarshal`
  The Ice run time invokes this method prior to marshaling the object's state, providing the opportunity for a subclass to validate its

declared data members.

- `ice_postUnmarshal`
  The Ice run time invokes this method after unmarshaling an object's state. A subclass typically overrides this method when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.

- `ice_dispatch`
  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.

Note that the generated class does not override `hashCode` and `equals`. This means that classes are compared using shallow reference equality, not value equality (as is used for structures).

All Slice classes derive from `Ice.Object` via the `Ice.ObjectImpl` abstract base class. `ObjectImpl` implements the `java.io.Serializable` interface to support Java's serialization facility. `ObjectImpl` also supplies an implementation of `clone` that returns a shallow member-wise copy.

Operations Interfaces

If a Slice class declares or inherits operations, or implements an interface, the Slice-to-Java translator will generate two additional interfaces named _<*class-name*>`Operations` and _<*class-name*>`OperationsNC`. The methods in the _<*class-name*>`Operations` interface have an additional trailing parameter of type `Ice.Current`, whereas the methods in the _<*class-name*>`OperationsNC` interface lack this additional trailing parameter. The methods without the `Current` parameter simply forward to the methods with a `Current` parameter, supplying a default `Current`. For now, you can ignore this parameter and pretend it does not exist.

Class Data Members in Java

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member. A JavaBean-style API is used for optional data members, and you can customize the mapping to force required members to use this same API.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    ["protected"] string tz;    // e.g. GMT, PST, EDT...
}
```

The Slice compiler produces the following generated code for this definition:

Java Compat

```
public class TimeOfDay extends ...
{
    protected short hour;
    protected short minute;
    protected short second;
    protected String tz;

    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second);
    // ...
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

You can optionally customize the mapping for data members to use getters and setters instead.

Class Operations in Java

Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

Operations of classes are mapped to abstract member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

Java Compat

```
public class FormattedTimeOfDayI extends FormattedTimeOfDay
{
    public String format(Ice.Current current)
    {
        DecimalFormat df = (DecimalFormat)DecimalFormat.getInstance();
        df.setMinimumIntegerDigits(2);
        return new String(df.format(hour) + ":" +
df.format(minute) + ":" + df.format(second));
    }
}
```

Value Factories in Java

> While value factories are necessary when using classes with operations (a now deprecated feature), value factories may be used for any kind of class and are *not* deprecated.

Having created a class such as `FormattedTimeOfDayI`, we have an implementation and we can instantiate the `FormattedTimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

Slice

```
interface Time
{
    FormattedTimeOfDay get();
}
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `FormattedTimeOfDay` class. However, `FormattedTimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `FormattedTimeOfDayI` class that implements the abstract `format` operation of the `FormattedTimeOfDay` abstr act class. In other words, we must provide the Ice run time with a factory that knows that the `FormattedTimeOfDay` abstract class has a `Fo rmattedTimeOfDayI` concrete implementation.

To supply the Ice run time with a value factory for our `FormattedTimeOfDayI` class, we must implement the `ValueFactory` interface:

Java Compat

```
class ValueFactory implements Ice.ValueFactory
{
    public Ice.Object create(String type)
    {
        assert(type.equals(FormattedTimeOfDay.ice_staticId()));
        return new FormattedTimeOfDayI();
    }
}
```

The factory's `create` method is called by the Ice run time when it needs to instantiate a `FormattedTimeOfDay` class.

The `create` method receives the type ID of the class to instantiate. For our `FormattedimeOfDay` class, the type ID is `"::Module::FormattedTimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `FormattedTimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoValueFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ValueFactory`, we must inform the Ice run time of the existence of the factory:

Java Compat

```
Ice.Communicator ic = ...;
ic.getValueFactoryManager().add(new ValueFactory(),
FormattedTimeOfDay.ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::Module::FormattedTimeOfDay"`, it calls the `create` method of our factory, which returns a `FormattedTimeOfDayI` instance to the Ice run time.

Finally, keep in mind that if a class has only data members, but no operations, you do not need to create and register a value factory to receive instances of such a class. You're only required to register a value factory when a class has operations.

Class Constructors in Java

Classes have a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default Java behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

The generated class also contains a second constructor that accepts one argument for each member of the class. This allows you to create and initialize a class in a single statement, for example:

Java Compat

```
TimeOfDay tod = new TimeOfDay(14, 45, 00, "PST"); // 14:45pm PST
```

For derived classes, the constructor requires an argument for every member of the class, including inherited members. For example, consider the the definition from Class Inheritance once more:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}

class DateTime extends TimeOfDay
{
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
}
```

The constructors for the generated classes are as follows:

Java Compat

```
public class TimeOfDay extends ...
{
    public TimeOfDay() {}

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // ...
}

public class DateTime extends TimeOfDay
{
    public DateTime() {}

    public DateTime(short hour, short minute, short second,
short day, short month, short year)
    {
        super(hour, minute, second);
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}
```

If you want to instantiate and initialize a `DateTime` instance, you must either use the default constructor or provide values for all of the data members of the instance, including data members of any base classes.

See Also

- Classes
- Class Inheritance
- Java Compat Mapping for Optional Data Members
- Type IDs
- Serializable Objects in Java Compat
- The Current Object
- Dispatch Interceptors
- Value Factories

Java Compat Mapping for Optional Data Members

The Java Compat mapping for optional data members in Slice classes and exceptions uses a JavaBean-style API that provides methods to get, set, and clear a member's value, and test whether a value is set. Consider the following Slice definition:

Slice

```
class C
{
    string name;
    optional(2) string alternateName;
    optional(5) bool active;
}
```

The generated Java code provides the following API:

Java Compat

```
public class C ...
{
    public C();
    public C(String name);
    public C(String name, String alternateName, boolean active);

    public String name;

    public String getAlternateName();
    public void setAlternateName(String v);
    public boolean hasAlternateName();
    public void clearAlternateName();
    public void optionalAlternateName(Ice.Optional<String> v);
    public Ice.Optional<String> optionalAlternateName();

    public boolean getActive();
    public void setActive(boolean v);
    public boolean isActive();
    public boolean hasActive();
    public void clearActive();
    public void optionalActive(Ice.BooleanOptional v);
    public Ice.BooleanOptional optionalActive();

    ...
}
```

If a class or exception declares any required data members, the generated class includes an overloaded constructor that accepts values for just the required members; optional members remain unset unless their Slice definitions specify a default value. Another overloaded constructor accepts values for all data members.

The `has` method allows you to test whether a member's value has been set, and the `clear` method removes any existing value for a member.

Calling a `get` method when the member's value has not been set raises `java.lang.IllegalStateException`.

The `optional` methods provide an alternate API that uses an `Optional` object to encapsulate the value, as discussed below.

Java Helper Classes for Optional Values

Ice defines the following classes to encapsulate optional values of primitive types:

- `Ice.BooleanOptional`
- `Ice.ByteOptional`
- `Ice.DoubleOptional`
- `Ice.FloatOptional`
- `Ice.IntOptional`
- `Ice.LongOptional`
- `Ice.ShortOptional`

These classes all share the same API for getting, setting, and testing an optional value. We'll use the `IntOptional` class as an example:

Java Compat

```
public class IntOptional
{
    public IntOptional();                // Value is unset
    public IntOptional(int v);           // Value is set
    public IntOptional(IntOptional opt); // Copies the state of the
argument
    public int get();                    // Raises IllegalStateException
if unset
    public void set(int v);              // Value is set
    public void set(IntOptional opt);    // Copies the state of the
argument
    public boolean isSet();
    public void clear();

    ...
}
```

The `Ice.Optional` generic class encapsulates values of reference types and offers an identical API:

Java Compat

```
public class Optional<T>
{
    public Optional();                // Value is unset
    public Optional(T v);             // Value is set
    public Optional(Optional<T> opt); // Copies the state of the
argument
    public T get();                   // Raises IllegalStateException if
unset
    public void set(T v);             // Value is set
    public void set(Optional<T> opt); // Copies the state of the
argument
    public boolean isSet();
    public void clear();

    public static <T> Optional<T> O(T v);
    public static BooleanOptional O(boolean v);
    public static ByteOptional O(byte v);
    public static ShortOptional O(short v);
    public static IntOptional O(int v);
    public static LongOptional O(long v);
    public static FloatOptional O(float v);
    public static DoubleOptional O(double v);

    ...
}
```

To improve the readability of code that creates optional values, the `Optional` class defines overloaded `O` methods that simplify the creation of these objects:

Java Compat

```
import static Ice.Optional.O;

C obj = ...;
Ice.Optional<C> opt = O(obj);

Ice.IntOptional i = O(5);
```

See Also

- Optional Data Members

Serializable Objects in Java Compat

In Java terminology, a *serializable object* typically refers to an object that implements the `java.io.Serializable` interface and therefore supports serialization to and from a byte stream. All Java classes generated from Slice definitions implement the `java.io.Serializable` interface.

In addition to serializing Slice types, applications may also need to incorporate foreign types into their Slice definitions. Ice allows you to pass Java serializable objects directly as operation parameters or as fields of another data type. For example:

Slice

```
["java:serializable:SomePackage.JavaClass"]
sequence<byte> JavaObj;
struct MyStruct
{
    int i;
    JavaObj o;
}

interface Example
{
    void op(JavaObj inObj, MyStruct s, out JavaObj outObj);
}
```

The generated code for `MyStruct` contains a member `i` of type `int` and a member `o` of type `SomePackage.JavaClass`:

Java Compat

```
public final class MyStruct implements java.lang.Cloneable
{
    public int i;
    public SomePackage.JavaClass o;

    // ...
}
```

The mapping for `op` uses `Ice.Holder<SomePackage.JavaClass>` for the out-parameter. (Out-parameters are always passed as `Ice.Holder<`*`class`*`>`.)

Java Compat

```
void op(SomePackage.JavaClass inObj,
        MyStruct s,
        Ice.Holder<SomePackage.JavaClass> outObj);
```

Of course, your client and server code must have an implementation of `JavaClass` that derives from `java.io.Serializable`:

Java Compat

```
package SomePackage;

public class JavaClass implements java.io.Serializable
{
    // ...
}
```

You can implement this class in any way you see fit — the Ice run time does not place any other requirements on the implementation.

See Also

- Serializable Objects

Customizing the Java Compat Mapping

You can customize the code that the Slice-to-Java compiler produces by annotating your Slice definitions with metadata. This section describes how metadata influences several aspects of the generated Java code.

On this page:

Java Packages

By default, the scope of a Slice definition determines the package of its mapped Java construct. A Slice type defined in a module hierarchy is mapped to a type residing in the equivalent Java package.

There are times when applications require greater control over the packaging of generated Java classes. For instance, a company may have software development guidelines that require all Java classes to reside in a designated package. One way to satisfy this requirement is to modify the Slice module hierarchy so that the generated code uses the required package by default. In the example below, we have enclosed the original definition of `Workflow::Document` in the modules `com::acme` so that the compiler will create the class in the `com.acme` package:

Slice

```
module com
{
    module acme
    {
        module Workflow
        {
            class Document
            {
                // ...
            }
        }
    }
}
```

There are two problems with this workaround:

1. It incorporates the requirements of an implementation language into the application's interface specification.
2. Developers using other languages, such as C++, are also affected.

The Slice-to-Java compiler provides a better way to control the packages of generated code through the use of metadata. The example above can be converted as follows:

Slice

```
["java:package:com.acme"]
module Workflow
{
    class Document
    {
        // ...
    }
}
```

The metadata directive `java:package:com.acme` instructs the compiler to generate all of the classes resulting from definitions in module `Workflow` into the Java package `com.acme`. The net effect is the same: the class for `Document` is generated in the package `com.acme.Workflow`. However, we have addressed the two shortcomings of the first solution by reducing our impact on the interface specification: the Slice-to-Java compiler recognizes the package metadata directive and modifies its actions accordingly, whereas the compilers for other language mappings simply ignore it.

> The `java:package` directive can also be applied as global metadata, in which case it serves as the default directive unless overridden by module metadata.

Java Package Configuration Properties

Using global metadata to alter the default package of generated classes has ramifications for the Ice run time when unmarshaling exceptions and concrete class types. The Ice run time dynamically loads generated classes by translating their Slice type IDs into Java class names. For example, the Ice run time translates the Slice type ID `::Workflow::Document` into the class name `Workflow.Document`.

However, when the generated classes are placed in a user-specified package, the Ice run time can no longer rely on the direct translation of a Slice type ID into a Java class name, and therefore requires additional configuration so that it can successfully locate the generated classes. Two configuration properties are supported:

- `Ice.Package.`*`Module=package`*
  Associates a top-level Slice module with the package in which it was generated.

  > Only top-level module names are allowed; the semantics of global metadata prevent a nested module from being generated into a different package than its enclosing module.

- `Ice.Default.Package=package`
  Specifies a default package to use if other attempts to load a class have failed.

The behavior of the Ice run time when unmarshaling an exception or concrete class is described below:

1. Translate the Slice type ID into a Java class name and attempt to load the class.
2. If that fails, extract the top-level module from the type ID and check for an `Ice.Package` property with a matching module name. If found, prepend the specified package to the class name and try to load the class again.
3. If that fails, check for the presence of `Ice.Default.Package`. If found, prepend the specified package to the class name and try to load the class again.
4. If the class still cannot be loaded, the instance may be sliced.

Continuing our example from the previous section, we can define the following property:

```
Ice.Package.Workflow=com.acme
```

Alternatively, we could achieve the same result with this property:

```
Ice.Default.Package=com.acme
```

Custom Types in Java

One of the more powerful applications of metadata is the ability to tailor the Java mapping for sequence and dictionary types to match the needs of your application.

Custom Type Metadata in Java

The metadata for specifying a custom type has the following format:

```
java:type:instance-type[:formal-type]
```

The formal type is optional; the compiler uses a default value if one is not defined. The instance type must satisfy an is-A relationship with the formal type: either the same class is specified for both types, or the instance type must be derived from the formal type.

The Slice-to-Java compiler generates code that uses the formal type for all occurrences of the modified Slice definition except when the generated code must instantiate the type, in which case the compiler uses the instance type instead.

The compiler performs no validation on your custom types. Misspellings and other errors will not be apparent until you compile the generated code.

Defining a Custom Sequence Type in Java

Although the default mapping of a sequence type to a native Java array is efficient and typesafe, it is not always the most convenient representation of your data. To use a different representation, specify the type information in a metadata directive, as shown in the following example:

Slice

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

It is your responsibility to use a type parameter for the Java class (`String` in the example above) that is the correct mapping for the sequence's element type.

The compiler requires the formal type to implement `java.util.List<`*E*`>`, where *E* is the Java mapping of the element type. If you do not specify a formal type, the compiler uses `java.util.List<`*E*`>` by default.

Note that extra care must be taken when defining custom types that contain nested generic types, such as a custom sequence whose element type is also a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Defining a Custom Dictionary Type in Java

The default instance type for a dictionary is `java.util.HashMap<`*K*`, `*V*`>`, where *K* is the Java mapping of the key type and *V* is the Java mapping of the value type. If the semantics of a `HashMap` are not suitable for your application, you can specify an alternate type using metadata as shown in the example below:

Slice

```
["java:type:java.util.TreeMap<String, String>"]
dictionary<string, string> StringMap;
```

It is your responsibility to use type parameters for the Java class (`String` in the example above) that are the correct mappings for the dictionary's key and value types.

The compiler requires the formal type to implement `java.util.Map<K, V>`. If you do not specify a formal type, the compiler uses this type by default.

Note that extra care must be taken when defining dictionary types that contain nested generic types, such as a dictionary whose element type is a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Using Custom Type Metadata in Java

You can define custom type metadata in a variety of situations. The simplest scenario is specifying the metadata at the point of definition:

Slice

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

Defined in this manner, the Slice-to-Java compiler uses `java.util.List<String>` (the default formal type) for all occurrences of `String List`, and `java.util.LinkedList<String>` when it needs to instantiate `StringList`.

You may also specify a custom type more selectively by defining metadata for a data member, parameter or return value. For instance, the mapping for the original Slice definition might be sufficient in most situations, but a different mapping is more convenient in particular cases. The example below demonstrates how to override the sequence mapping for the data member of a structure as well as for several operations:

Slice

```
sequence<string> StringSeq;

struct S
{
    ["java:type:java.util.LinkedList<String>"] StringSeq seq;
}

interface I
{
    ["java:type:java.util.ArrayList<String>"] StringSeq
    modifiedReturnValue();

    void modifiedInParam(["java:type:java.util.ArrayList<String>"] Stri
ngSeq seq);

    void modifiedOutParam(out ["java:type:java.util.ArrayList<String>"]
StringSeq seq);
}
```

As you might expect, modifying the mapping for an operation's parameters or return value may require the application to manually convert values from the original mapping to the modified mapping. For example, suppose we want to invoke the `modifiedInParam` operation. The signature of its proxy operation is shown below:

Java Compat

```
void modifiedInParam(java.util.List<String> seq)
```

The metadata changes the mapping of the `seq` parameter to `java.util.List`, which is the default formal type. If a caller has a `StringSeq` value in the original mapping, it must convert the array as shown in the following example:

Java Compat

```
String[] seq = new String[2];
seq[0] = "hi";
seq[1] = "there";
IPrx proxy = ...;
proxy.modifiedInParam(java.util.Arrays.asList(seq));
```

Although we specified the instance type `java.util.ArrayList<String>` for the parameter, we are still able to pass the result of `asList` because its return type (`java.util.List<String>`) is compatible with the parameter's formal type declared by the proxy method. In the case of an operation parameter, the instance type is only relevant to a servant implementation, which may need to make assumptions about the actual type of the parameter.

Mapping for Modified Out Parameters in Java

The mapping for an `out` parameter uses a generated "holder" class to convey the parameter value. If you modify the mapping of an `out` parameter, as discussed in the previous section, it is possible that the holder class for the parameter's unmodified type is no longer compatible with the custom type you have specified. The holder class generated for `StringSeq` is shown below:

Java Compat

```
public final class StringSeqHolder
{
    public StringSeqHolder() {}

    public StringSeqHolder(String[] value)
    {
        this.value = value;
    }

    public String[] value;
}
```

An `out` parameter of type `StringSeq` would normally map to a proxy method that used `StringSeqHolder` to hold the parameter value. When the parameter is modified, as is the case with the `modifiedOutParam` operation, the Slice-to-Java compiler cannot use `StringSeqHolder` to hold an instance of `java.util.List<String>`, because `StringSeqHolder` is only appropriate for the default mapping to a native array.

As a result, the compiler handles these situations using instances of the generic class `Ice.Holder<`*T*`>`, where *T* is the parameter's formal type. Consider the following example:

Slice

```
sequence<string> StringSeq;

interface I
{
    void modifiedOutParam(out ["java:type:java.util.ArrayList<String>"]
StringSeq seq);
}
```

The compiler generates the following mapping for the `modifiedOutParam` proxy method:

Java Compat

```
void modifiedOutParam(Ice.Holder<java.util.List<java.lang.String>> seq)
```

The formal type of the parameter is `java.util.List<String>`, therefore the holder class becomes `Ice.Holder<java.util.List<String>>`.

Buffer Types in Java

You can annotate sequences of certain primitive types with the `java:buffer` metadata tag to change the mapping to use subclasses of `java.nio.Buffer`. This mapping provides several benefits:

- You can pass a buffer to a Slice API instead of creating and filling a temporary array
- If you need to pass a portion of an existing array, you can wrap it with a buffer and avoid an extra copy
- Receiving buffers during a Slice operation also avoids copying by directly referencing the data in Ice's unmarshaling buffer

> To use buffers safely, applications must disable caching by setting `Ice.CacheMessageBuffers` to zero.

The following table lists each supported Slice primitive type with its corresponding mapped class:

Primitive	Mapping
`byte`	`java.nio.ByteBuffer`
`short`	`java.nio.ShortBuffer`
`int`	`java.nio.IntBuffer`
`long`	`java.nio.LongBuffer`
`float`	`java.nio.FloatBuffer`
double	`java.nio.DoubleBuffer`

The `java:buffer` tag can be applied to the initial definition of a sequence, in which case the mapping uses the buffer type for all occurrences of that sequence type:

Slice

```
["java:buffer"] sequence<int> Values;

struct Observation
{
    int x;
    int y;
    Values measurements;
}
```

We can construct an `Observation` as follows:

Java Compat

```
Observation obs = new Observation();
obs.x = 5;
obs.y = 9;
obs.measurements = java.nio.IntBuffer.allocate(10);
for(int i = 0; i < obs.measurements.capacity(); ++i)
{
    obs.measurements.put(i, ...);
}
```

The `java:buffer` tag can also be applied in more limited situations to override a sequence's normal mapping:

Slice

```
sequence<byte> ByteSeq; // Maps to byte[]

struct Page
{
    int offset;
    ["java:buffer"] ByteSeq data; // Maps to java.nio.ByteBuffer
}

interface Decoder
{
    ["java:buffer"] ByteSeq decode(ByteSeq data);
```

In this example, `ByteSeq` maps by default to a `byte` array, but we've overridden the mapping to use a buffer when this type is used as a data member in `Page` and as the return value of the `decode` operation; the input parameter to `decode` uses the default array mapping.

JavaBean Mapping

The Java mapping optionally generates JavaBean-style methods for the data members of class, structure, and exception types.

JavaBean Generated Methods

For each data member *val* of type *T*, the mapping generates the following methods:

Java Compat

```
public T getVal();
public void setVal(T v);
```

The mapping generates an additional method if *T* is the `bool` type:

Java Compat

```
public boolean isVal();
```

Finally, if *T* is a sequence type with an element type *E*, two methods are generated to provide direct access to elements:

Java Compat

```
public E getVal(int index);
public void setVal(int index, E v);
```

Note that these element methods are only generated for sequence types that use the default mapping.

The Slice-to-Java compiler considers it a fatal error for a JavaBean method of a class data member to conflict with a declared operation of the class. In this situation, you must rename the operation or the data member, or disable the generation of JavaBean methods for the data member in question.

JavaBean Metadata

The JavaBean methods are generated for a data member when the member or its enclosing type is annotated with the `java:getset` meta data. The following example demonstrates both styles of usage:

Slice

```
sequence<int> IntSeq;

class C
{
    ["java:getset"] int i;
    double d;
}

["java:getset"]
struct S
{
    bool b;
    string str;
}

["java:getset"]
exception E
{
    IntSeq seq;
}
```

JavaBean methods are generated for all members of struct S and exception E, but for only one member of class C. Relevant portions of the generated code are shown below:

Java Compat

```
public class C extends ...
{
    ...

    public int i;

    public int getI()
    {
        return i;
    }

    public void setI(int _i)
    {
        i = _i;
    }

    public double d;
}

public final class S implements java.lang.Cloneable
{
```

```
    public boolean b;

    public boolean getB()
    {
        return b;
    }

    public void setB(boolean _b)
    {
        b = _b;
    }

    public boolean isB()
    {
        return b;
    }

    public String str;

    public String getStr()
    {
        return str;
    }

    public void setStr(String _str)
    {
        str = _str;
    }

    ...
}

public class E extends UserException
{
    ...

    public int[] seq;

    public int[] getSeq()
    {
        return seq;
    }

    public void setSeq(int[] _seq)
    {
        seq = _seq;
    }

    public int getSeq(int _index)
    {
```

```
        return seq[_index];
    }

    public void setSeq(int _index, int _val)
    {
        seq[_index] = _val;
    }
```

```
    ...
}
```

Overriding `serialVersionUID`

The Slice-to-Java compiler computes a default value for the `serialVersionUID` member of Slice classes, exceptions and structures. If you prefer, you can override this value using the `java:serialVersionUID` metadata, as shown below:

Slice

```
["java:serialVersionUID:571254925"]
struct Identity
{
    ...
}
```

The specified value will be used in place of the default value in the generated code:

Java Compat

```
public class Identity
{
    ...

    public static final long serialVersionUID = 571254925L;
}
```

By using this metadata, the application assumes responsibility for updating the UID whenever changes to the Slice definition affect the serializable state of the type.

See Also

- Metadata
- Java Compat Mapping for Modules
- Java Compat Mapping for Operations
- Class Inheritance Semantics

Asynchronous Method Invocation (AMI) in Java Compat

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

Basic Asynchronous API in Java

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in Java

Besides the synchronous proxy methods, `slice2java` generates the following asynchronous proxy methods:

Java Compat

```
public interface EmployeesPrx extends Ice.ObjectPrx
{
    // ...

    public Ice.AsyncResult begin_getName(int number);
    public Ice.AsyncResult begin_getName(int number,
java.util.Map<String, String> __ctx);

    public String end_getName(Ice.AsyncResult __result);
}
```

Four additional overloads of `begin_getName` are generated for use with generic completion callbacks and type-safe completion callbacks.

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. (The `begin_` method is overloaded so you can pass a per-invocation context.)

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

Java Compat

```
EmployeesPrx e = ...;
Ice.AsyncResult r = e.begin_getName(99);

// Continue to do other things here...

String name = e.end_getName(r);
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `AsyncResult`. This value contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResult` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. Similarly, the `end_` method has one out-parameter for each out-parameter of the corresponding Slice operation (plus the `AsyncResult` parameter). For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

Java Compat

```
Ice.AsyncResult begin_op(int inp1, String inp2);
Ice.AsyncResult begin_op(int inp1, String inp2,
java.util.Map<String, String> __ctx);
double end_op(Ice.BooleanHolder outp1, Ice.LongHolder outp2,
Ice.AsyncResult r);
```

Asynchronous Exception Semantics in Java

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `java.lang.IllegalArgumentException`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

`AsyncResult` Interface in Java

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

Java Compat

```
public interface AsyncResult
{
    public void cancel();

    public Communicator getCommunicator();
    public Connection getConnection();
    public ObjectPrx getProxy();
    public String getOperation();

    public boolean isCompleted();
    public void waitForCompleted();

    public boolean isSent();
    public void waitForSent();

    public void throwLocalException();

    public boolean sentSynchronously();
}
```

The methods have the following semantics:

- `void cancel()`
  This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. `cancel` is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning `isCompleted` returns true, and the result of the invocation is an `Ice.InvocationCanceledException`.
- `Communicator getCommunicator()`
  This method returns the communicator that sent the invocation.
- `Connection getConnection()`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `getConnection` method only returns a non-nil value when the `AsyncResult` object is obtained by calling `begin_flushBatchRequests` on a `Connection` object.
- `ObjectPrx getProxy()`
  This method returns the proxy that was used to call the `begin_` method, or nil if the `AsyncResult` object was not obtained via an asynchronous proxy invocation.
- `String getOperation()`
  This method returns the name of the operation.
- `boolean isCompleted()`
  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.
- `void waitForCompleted()`
  This method blocks the caller until the result of an invocation becomes available.
- `boolean isSent()`
  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.
- `void waitForSent()`

This method blocks the calling thread until a request has been written to the client-side transport, or an exception occurs. After `waitForSent` returns, `isSent` returns true if the request was successfully written to the client-side transport, or false if an exception occurred. In the case of a failure, you can call the corresponding `end_` method or `throwLocalException` to obtain the exception.

- `void throwLocalException()`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.
- `boolean sentSynchronously()`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

Polling for Completion in Java

The `AsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

Java Compat

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;
while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

Java Compat

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;

LinkedList<Ice.AsyncResult> results = new LinkedList<Ice.AsyncResult>();
int numRequests = 5;

while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice.AsyncResult r = ft.begin_send(offset, bs);
    offset += bs.length;

    // Wait until this request has been passed to the transport.
    r.waitForSent();
    results.add(r);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.size() > numRequests)
    {
        Ice.AsyncResult r = results.getFirst();
        results.removeFirst();
        r.waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while(results.size() > 0)
{
    Ice.AsyncResult r = results.getFirst();
    results.removeFirst();
    r.waitForCompleted();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Generic Completion Callbacks in Java

The `begin_` method is overloaded to allow you to provide completion callbacks. Here are the corresponding methods for the `getName` operation:

Java Compat

```
Ice.AsyncResult begin_getName(int number, Ice.Callback __cb);

Ice.AsyncResult begin_getName(int number,
                              java.util.Map<String, String> __ctx,
                              Ice.Callback __cb);
```

The second version of `begin_getName` lets you override the default context. Following the in-parameters, the `begin_` method accepts a parameter of type `Ice.Callback`, which is a callback class with a `completed` method that you must provide. The Ice run time invokes the `completed` method when an asynchronous operation completes. For example:

Java Compat

```
public class MyCallback extends Ice.Callback
{
    public void completed(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try
        {
            String name = e.end_getName(r);
            System.out.println("Name is: " + name);
        }
        catch(Ice.LocalException ex)
        {
            System.err.println("Exception is: " + ex);
        }
    }
}
```

Note that your callback class must derive from `Ice.Callback`. The implementation of your callback method must call the `end_` method. The proxy for the call is available via the `getProxy` method on the `AsyncResult` that is passed by the Ice run time. The return type of `getProxy` is `Ice.ObjectPrx`, so you must down-cast the proxy to its correct type.

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. If an operation can throw user exceptions, this means that you need an additional catch handler for `Ice.UserException` (or catch all possible user exceptions explicitly). If you allow an exception to escape from the callback method, the Ice run time produces a log entry by default and ignores the exception. (You can disable the log message by setting the property `Ice.Warn.AMICallback` to zero.)

To inform the Ice run time that you want to receive a callback for the completion of the asynchronous call, you pass the callback instance to the `begin_` method:

Java Compat

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb);
```

This is often written using an anonymous class instead:

Java Compat

```
EmployeesPrx e = ...;

e.begin_getName(
        99,
        new Ice.Callback()
        {
            public void completed(Ice.AsyncResult r)
            {
                EmployeesPrx p = (EmployeesPrx)r.getProxy();
                try
                {
                    String name = p.end_getName(r);
                    System.out.println("Name is: " + name);
                }
                catch(Ice.LocalException ex)
                {
                    System.err.println("Exception: " + ex);
                }
            }
        });
```

An anonymous class is especially useful for callbacks that do only a small amount of work because the code that starts the call and the code that processes the results are physically close together.

Sharing State Between `begin_` and `end_` Methods in Java

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

Assuming that we have a `Widget` class that designates a particular user interface element, you could pass different widgets by storing the widget to be used as a member of your callback class:

Java Compat

```
public class MyCallback extends Ice.Callback
{
    public MyCallback(Widget w)
    {
        _w = w;
    }

    private Widget _w;

    public void completed(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try
        {
            String name = e.end_getName(r);
            _w.writeString(name);
        }
        catch(Ice.LocalException ex)
        {
            System.err.println("Exception is: " + ex);
        }
    }
}
```

For this example, we assume that widgets have a `writeString` method that updates the relevant UI element.

When you call the `begin_` method, you pass the appropriate callback instance to inform the `end_` method how to update the display:

Java Compat

```
EmployeesPrx e = ...;
Widget widget1 = ...;
Widget widget2 = ...;

// Invoke the getName operation with different widget callbacks.
e.begin_getName(99, new MyCallback(widget1));
e.begin_getName(24, new MyCallback(widget2));
```

The callback class provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same callback instance to multiple invocations. (If you do this, your callback methods may need to use synchronization.)

Type-Safe Completion Callbacks in Java

The generic callback API is not entirely type-safe:

- You must down-cast the return value of `getProxy` to the correct proxy type before you can call the `end_` method.

- You must call the correct `end_` method to match the operation called by the `begin_` method.
- You must remember to catch exceptions when you call the `end_` method; if you forget to do this, you will not know that the operation failed.

`slice2java` generates an additional type-safe API that takes care of these chores for you. To use type-safe callbacks, you can either implement a callback class or use lambda functions.

Type-Safe Callback Classes in Java

A callback class must define two callback methods:

- a `response` method that is called if the operation succeeds
- an `exception` method that is called if the operation raises an exception

The class must derive from the base class that is generated by `slice2java`. The name of this base class is `<module>.Callback_<interface>_<operation>`. Here is a callback class for an invocation of the `getName` operation:

Java Compat

```
public class MyCallback extends Demo.Callback_Employees_getName
{
    public void response(String name)
    {
        System.out.println("Name is: " + name);
    }

    public void exception(Ice.LocalException ex)
    {
        System.err.println("Exception is: " + ex);
    }
}
```

The `response` callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the `response` callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

The `exception` callback is invoked if the invocation fails because of an Ice run time exception. If the Slice operation can also raise user exceptions, your callback class must supply an additional overloading of `exception` that accepts an argument of type `Ice.UserException`.

The proxy methods are overloaded to accept this callback instance:

Java Compat

```
Ice.AsyncResult begin_getName(int number,
                              Callback_Employees_getName __cb);

Ice.AsyncResult begin_getName(int number,
                              java.util.Map<String, String> __ctx,
                              Callback_Employees_getName __cb);
```

You pass the callback to an invocation as you would with the generic API:

Java Compat

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb);
```

Type-Safe Lambda Functions in Java

You can implement your type-safe callbacks using in-line lambda functions. The proxy methods are overloaded to accept these functions:

Java Compat

```
Ice.AsyncResult begin_getName(int number,

IceInternal.Functional_GenericCallback1<String> __responseCb,

IceInternal.Functional_GenericCallback1<Ice.Exception> __exceptionCb);

Ice.AsyncResult begin_getName(int number,
                              java.util.Map<String, String> __ctx,

IceInternal.Functional_GenericCallback1<String> __responseCb,

IceInternal.Functional_GenericCallback1<Ice.Exception> __exceptionCb);
```

The names of the internal interfaces used in the API are not important; what matters are their parameterized types, which tell you the arguments that your lambda functions must accept. For example, we can call `begin_getName` as follows:

Java Compat

```
EmployeesPrx e = ...;

e.begin_getName(99,
    (String name) ->
        {
            System.out.println("Name is: " + name);
        },
    (Ice.Exception ex) ->
        {
            System.err.println("Exception is: " + ex);
        });
```

Asynchronous Oneway Invocations in Java

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws an `IllegalArgumentException`.

The callback methods looks exactly as for a twoway invocation. For the generic API, the Ice run time does not call the `completed` callback method unless the invocation raised an exception during the `begin_` method ("on the way out"). For the type-safe API, the `response` meth od is never called.

Flow Control in Java

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

For the generic API, you can override the `sent` method:

Java Compat

```
public class MyCallback extends Ice.Callback
{
    public void completed(Ice.AsyncResult r)
    {
        // ...
    }

    public void sent(Ice.AsyncResult r)
    {
        // ...
    }
}
```

You inform the Ice run time that you want to be informed when a call has been passed to the local transport as usual:

Java Compat

```
e.begin_getName(99, new MyCallback());
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the `sent` method from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the `sent` method from a different thread once it has written the request to the local transport. In addition, you can find out from the `AsyncResult` that is returned by the `begin_` method whether the request was sent synchronously or was queued, by calling `sentSynchronously`.

For the generic API, the `sent` method has the following signature:

Java Compat

```
void sent(Ice.AsyncResult r);
```

For the type-safe API, the signature is:

Java Compat

```
void sent(boolean sentSynchronously);
```

For the generic API, you can find out whether the request was sent synchronously by calling `sentSynchronously` on the `AsyncResult`. For the type-safe API, the boolean `sentSynchronously` parameter provides the same information.

The `sent` methods allow you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Asynchronous Batch Requests in Java

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws an `IllegalArgumentException`.

A batch oneway invocation never calls the generic or type-safe callbacks unless an error occurs before the request is queued. The returned `Ice.AsyncResult` for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The returned result can also be marked completed if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency Semantics for AMI in Java

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or parameter.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch

slice2java Command-Line Options (Java Compat)

The Slice-to-Java compiler, `slice2java`, generates code for both the Java and Java Compat mappings, and is described on this page.

See Also

- Using the Slice Compilers

Using Slice Checksums in Java Compat

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2java`, the `--checksum` option causes the compiler to generate a new Java class that adds checksums to a static map member. Assuming we supplied the option `--checksum Checksums` to `slice2java`, the generated class `Checksums.java` looks like this:

Java Compat

```
public class Checksums
{
    public static java.util.Map<String, String> checksums;
}
```

The read-only map `checksums` is initialized automatically prior to first use; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the dictionaries using the `equals` method. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

Java Compat

```
java.util.Map<String, String> serverChecksums = ...
Checksums.checksums.forEach((id, checksum) ->
    {
        String serverChecksum = serverChecksums.get(id);
        if(serverChecksum == null)
        {
            // No match found for type id!
        }
        else if(!checksum.equals(serverChecksum))
        {
            // Checksum mismatch!
        }
    });
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums

Example of a File System Client in Java Compat

This page presents the source code for a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The Java code hardly differs from the code you would write for an ordinary Java program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Java object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side Java mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

Java Compat

```
import Filesystem.*;

public class Client
{
```

```
    // Recursively print the contents of directory "dir" in
    // tree fashion.  For files, show the contents of each file.
    // The "depth" parameter is the current nesting level
    // (for indentation).

    static void listRecursive(DirectoryPrx dir, int depth)
    {
        char[] indentCh = new char[++depth];
        java.util.Arrays.fill(indentCh, '\t');
        String indent = new String(indentCh);

        NodePrx[] contents = dir.list();

        for(int i = 0; i < contents.length; ++i)
        {
            DirectoryPrx subdir
= DirectoryPrxHelper.checkedCast(contents[i]);
            FilePrx file = FilePrxHelper.uncheckedCast(contents[i]);
            System.out.println(indent + contents[i].name() +
                (subdir != null ? " (directory):" : " (file):"));
            if(subdir != null)
            {
                listRecursive(subdir, depth);
            }
            else
            {
                String[] text = file.read();
                for(int j = 0; j < text.length; ++j)
                {
                    System.out.println(indent + "\t" + text[j]);
                }
            }
        }
    }

    public static void main(String[] args) throws Exception
    {
        try(com.zeroc.Ice.Communicator ic = Ice.Util.initialize(args))
        {
            //
            // Create a proxy for the root directory
            //
            Ice.ObjectPrx base
= ic.stringToProxy("RootDir:default -p 10000");
            if(base == null)
            {
                throw new RuntimeException("Cannot create proxy");
            }

            //
```

```
        // Down-cast the proxy to a Directory proxy
        //
        DirectoryPrx rootDir = DirectoryPrxHelper.checkedCast(base);
        if(rootDir == null)
        {
            throw new RuntimeException("Invalid proxy");
        }

        //
        // Recursively list the contents of the root directory
        //
        System.out.println("Contents of root directory:");
        listRecursive(rootDir, 0);
    }
```

```
        }
    }
```

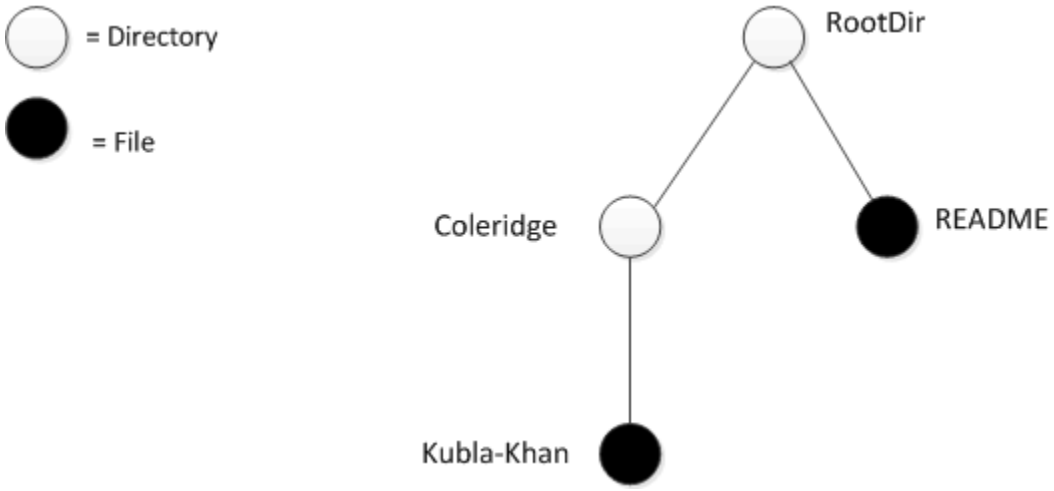
After importing the `Filesystem` package, the `Client` class defines two methods: `listRecursive`, which is a helper function to print the contents of the file system, and `main`, which is the main program. Let us look at `main` first:

1. The structure of the code in `main` follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Example of a File System Server in Java Compat
- Object Life Cycle
- IceGrid

Server-Side Slice-to-Java Compat Mapping

The mapping for Slice data types to Java is identical on the client side and server side. This means that everything in Client-Side Slice-to-Java Compat Mapping also applies to the server side. However, for the server side, there are a few additional things you need to know — specifically how to:

- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time

Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Although the examples we present are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Java mapping rules beyond those we described here.

Topics

Server-Side Java Compat Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing member functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Types in Java

On the client side, interfaces map to proxy types. On the server side, interfaces map to *skeleton* types. A skeleton is a class or interface that defines a method for each operation on the corresponding Slice interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The Slice compiler generates the following definition for this interface:

Java Compat

```
package Filesystem;

public interface _NodeOperations
{
    String name(Ice.Current current);
}

public interface _NodeOperationsNC
{
    String name();
}

public interface Node extends Ice.Object,
_NodeOperations, _NodeOperationsNC
{
    public static final String ice_staticId = "::Filesystem::Node";
    public static final long serialVersionUID = ...;
}

public abstract class _NodeDisp extends Ice.ObjectImpl implements Node
{
    // Mapping-internal code here...
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to Java packages with the same name, so the skeleton class definitions are part of the `Filesystem` package.
- For each Slice interface <*`interface-name`*>, the compiler generates Java interfaces `_`<*`interface-name`*>`Operations` and `_`<*`interface-name`*>`OperationsNC` (`_NodeOperations` and `_NodeOperationsNC` in this example). These interfaces contain a method for each operation in the Slice interface. (You can ignore the `Ice.Current` parameter for now.)
- For each Slice interface <*`interface-name`*>, the compiler generates a Java interface <*`interface-name`*> (`Node` in this example). That interface extends `Ice.Object` as well as the two operations interfaces, and defines the constant `ice_staticId` with the corresponding Slice type ID.
- For each Slice interface <*`interface-name`*>, the compiler generates an abstract class `_`<*`interface-name`*>`Disp` (`_NodeDisp` in this example). This abstract class is the actual skeleton class; it is the base class from which you derive your servant class.

Servant Classes in Java

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton type. For example, to create a servant for the `Node` interface, you could write:

Java Compat

```
package Filesystem;

public final class NodeI extends _NodeDisp
{
    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.)

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` function returns its value.)

Normal and `idempotent` Operations in Java

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

Slice

```
interface Example
{
                void   normalOp();
    idempotent void   idempotentOp();
    idempotent string readonlyOp();
}
```

The methods for this interface look like this:

Java Compat

```
void normalOp(Current current);
void idempotentOp(Current current);
String readonlyOp(Current current);
```

Note that the signatures of the member functions are unaffected by the `idempotent` qualifier.

See Also

- Slice for a Simple File System
- Java Compat Mapping for Interfaces
- Parameter Passing in Java Compat
- Raising Exceptions in Java Compat
- Tie Classes in Java Compat
- The Current Object

Parameter Passing in Java Compat

For each parameter of a Slice operation, the Java mapping generates a corresponding parameter for the method in the `_<interface-name>Operations` interface. Additionally, every operation receives a trailing parameter of type `Current`. For example, the `name` operation of the `Node` interface has no parameters, but the corresponding `name` method of the servant interface has a single parameter of type `Current`. We will ignore this parameter for now.

> The parameter-passing rules change somewhat when using the asynchronous mapping.

On this page:

Passing Required Parameters in Java

Parameter passing on the server side follows the rules for the client side. To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
module M
{
    interface Example
    {
        string op(string sin, out string sout);
    }
}
```

The generated skeleton class for this interface looks as follows:

Java Compat

```
public interface _ExampleOperations
{
    String op(String sin, Ice.StringHolder sout, Ice.Current current);
}
```

As you can see, there are no surprises here. For example, we could implement op as follows:

Java Compat

```
public final class ExampleI extends M._ExampleDisp
{
    public String op(String sin, Ice.StringHolder sout,
Ice.Current current)
    {
        System.out.println(sin);     // In params are initialized
        sout.value = "Hello World!"; // Assign out param
        return "Done";
    }
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal Java rules and do not require special-purpose API calls.

Passing Optional Parameters in Java

Suppose we modify the example above to use optional parameters:

Slice

```
module M
{
    interface Example
    {
        optional(1) string op(optional(2) string sin, out optional(3)
string sout);
    }
}
```

The generated skeleton now looks like this:

Java Compat

```
public interface _ExampleOperations
{
    String op(Ice.Optional<String> sin, Ice.StringHolder sout,
Ice.Current current);
}
```

The default mapping treats optional out parameters and return values as if they are required. If your servant needs the ability to return an optional value, you must add the `java:optional` metadata tag:

Slice

```
module M
{
    interface Example
    {
        ["java:optional"]
        optional(1) string op(optional(2) string sin, out optional(3)
string sout);
    }
}
```

This tag can be applied to an interface if you want to use the optional mapping for all of the operations in that interface, or to individual operations as shown here. With this change, the mapping now returns optional values:

Java Compat

```
public interface _ExampleOperations
{
    Ice.Optional<String> op(Ice.Optional<String> sin,
Ice.Optional<String> sout, Ice.Current current);
}
```

The `java:optional` tag affects the return value and all out parameters; it is not possible to modify the mapping only for certain parameters.

Thread-Safe Marshaling in Java

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For Java applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following servant implementation:

Java

```
public class GridI extends _GridDisp
{
    GridI()
    {
        _grid = // ...
    }

    public int[][] getGrid(Current current)
    {
        return _grid;
    }

    public void setValue(int x, int y, int val, Current current)
    {
        _grid[x][y] = val;
    }

    private int[][] _grid;
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned array in preparation to send a reply message, it is possible for another thread to dispatch the `setValue` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

Java

```
public class GridI implements Grid
{
    ...

    public synchronized int[][] getGrid(Current current)
    {
        return _grid;
    }

    public synchronized void
setValue(int x, int y, int val, Current current)
    {
        int[][] newGrid = // shallow copy...
        newGrid[x][y] = val;
        _grid = newGrid;
    }

    ...
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach changes accessor operations to use AMD in order to regain control over marshaling. After annotating the `getGrid` operation with `amd` metadata, we can revise the servant as follows:

Java

```
public class GridI extends _GridDisp
{
    ...

    public synchronized void
getGrid_async(AMD_Grid_getGrid cb, Current current)
    {
        cb.ice_response(_grid);
    }

    public synchronized void
setValue(int x, int y, int val, Current current)
    {
        _grid[x][y] = val;
    }

    ...
}
```

Normally, AMD is used in situations where the servant needs to delay its response to the client without blocking the calling thread. For `getGrid`, that is not the goal; instead, as a side-effect, AMD provides the desired marshaling behavior. Specifically, the Ice run time marshals the reply to an asynchronous request at the time the servant invokes `ice_response` on the AMD callback object. Because `getGrid` and `setValue` are synchronized, this guarantees that the data remains in a consistent state during marshaling.

See Also

- Server-Side Java Compat Mapping for Interfaces
- Java Compat Mapping for Operations
- Raising Exceptions in Java Compat
- Tie Classes in Java Compat
- The Current Object

Raising Exceptions in Java Compat

Servant Exceptions in Java

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

Java Compat

```
@Override
public void write(String[] text, Current current)
    throws GenericError
{
    try
    {
        // Try to write file contents here...
    }
    catch(Exception ex)
    {
        throw new GenericError("Exception during write operation", ex);
    }
}
```

Note that, for this example, we have supplied the optional second parameter to the `GenericError` constructor. This parameter sets the inner exception and preserves the original cause of the error for later diagnosis.

If you throw an arbitrary Java run-time exception (such as a `ClassCastException`), the Ice run time catches the exception and then returns an `UnknownException` to the client.

Tthe server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw an Ice run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

JVM Error Semantics

Servant implementations might inadvertently raise low-level errors during the course of their operation. These exceptions, which are subclasses of `java.lang.Error`, should not normally be trapped by the servant because they often indicate the occurrence of a serious, unrecoverable situation. For example, the JVM throws `StackOverflowError` when a recursive method has used all available stack space.

The Ice run time traps instances of `java.lang.Error` thrown by servants and then attempts to log the exception and send an `UnknownException` to the client. This attempt may or may not succeed, depending on the nature of the error and the condition of the JVM. As an example, the servant implementation might raise `OutOfMemoryError` and Ice's attempt to log the error and send a response could also fail due to lack of memory.

For an occurrence of `OutOfMemoryError` or `AssertionError`, Ice does not re-throw the error after logging a message and sending a response. For all other subclasses of `Error`, Ice re-throws the error so that the JVM's normal error-handling strategy will execute.

> When the JVM raises a subclass of `Error`, it usually means that a significant problem has occurred. Ice tries to send an `UnknownException` to the client (for twoway requests) in order to prevent the client from waiting indefinitely for a response. However, the

JVM may prevent Ice from successfully sending this response, which is another reason your clients should use invocation timeouts as a defensive strategy. Finally, in nearly all occurrences of an error, the server is unlikely to continue operating correctly even if Ice is able to complete the client's request. For example, the JVM terminates the thread that raised an uncaught error. In the case of an uncaught error raised by a servant, the thread being terminated is normally from the Ice server-side thread pool. If all of the threads in this pool eventually terminate due to uncaught errors, the server can no longer respond to new client requests.

See Also

- Run-Time Exceptions
- Java Compat Mapping for Exceptions
- Server-Side Java Compat Mapping for Interfaces
- Parameter Passing in Java Compat
- Tie Classes in Java Compat

Tie Classes in Java Compat

The Java Compat mapping to skeleton classes requires the servant class to inherit from its skeleton class. Occasionally, this creates a problem: some class libraries require you to inherit from a base class in order to access functionality provided by the library; because Java does not support multiple implementation inheritance, this means that you cannot use such a class library to implement your servants because your servants cannot inherit from both the library class and the skeleton class simultaneously.

To allow you to still use such class libraries, Ice provides a way to write servants that replaces inheritance with delegation. This approach is supported by *tie classes*. The idea is that, instead of inheriting from the skeleton class, you simply create a class (known as an *implementation class* or *delegate class*) that contains methods corresponding to the operations of an interface. You use the `["java:tie"]` metadata directive to create a tie class. For example, adding this directive in front of the `Node` interface we saw previously makes `slice2java` emit an additional tie class. For an interface `<interface-name>`, the generated tie class has the name `_<interface-name>Tie`:

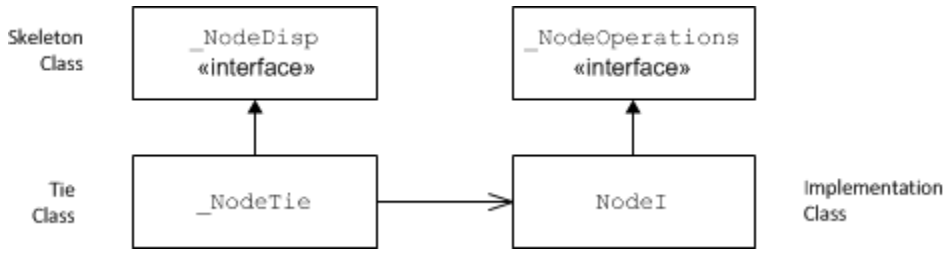
Slice

```
module Filesystem
{
    ["java:tie"] interface Node
    {
      ...
   }
}
```

Java Compat

```
package Filesystem;

public class _NodeTie extends _NodeDisp implements Ice.TieBase
{
    public _NodeTie() {}

    public _NodeTie(_NodeOperations delegate)
    {
        _ice_delegate = delegate;
    }

    public java.lang.Object ice_delegate()
    {
        return _ice_delegate;
    }

    public void ice_delegate(java.lang.Object delegate)
    {
        _ice_delegate = (_NodeOperations)delegate;
    }

    public boolean equals(java.lang.Object rhs)
    {
        if(this == rhs)
        {
            return true;
        }
        if(!(rhs instanceof _NodeTie))
        {
            return false;
        }

        return _ice_delegate.equals(((_NodeTie)rhs)._ice_delegate);
    }

    public int hashCode()
    {
        return _ice_delegate.hashCode();
    }

    public String name(Ice.Current current)
    {
        return _ice_delegate.name(current);
    }

    private _NodeOperations _ice_delegate;
}
```

This looks a lot worse than it is: in essence, the generated tie class is simply a servant class (it extends `_NodeDisp`) that delegates to your implementation class each invocation of a method corresponding to a Slice operation:



A skeleton class, tie class, and implementation class.

The generated tie class also implements the `Ice.TieBase` interface, which defines methods for obtaining and changing the delegate object:

Java Compat

```
package Ice;

public interface TieBase
{
    java.lang.Object ice_delegate();
    void ice_delegate(java.lang.Object delegate);
}
```

The delegate has type `java.lang.Object` in these methods in order to allow a tie object's delegate to be manipulated without knowing its actual type. However, the `ice_delegate` modifier raises `ClassCastException` if the given delegate object is not of the correct type.

Given this machinery, we can create an implementation class for our `Node` interface as follows:

Java Compat

```
package Filesystem;

public final class NodeI implements _NodeOperations
{
    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
```

Note that this class is identical to our previous implementation, except that it implements the `_NodeOperations` interface and does not extend `_NodeDisp` (which means that you are now free to extend any other class to support your implementation).

To create a servant, you instantiate your implementation class and the tie class, passing a reference to the implementation instance to the tie constructor:

Java Compat

```
NodeI fred = new NodeI("Fred");         // Create implementation
_NodeTie servant = new _NodeTie(fred);  // Create tie
```

Alternatively, you can also default-construct the tie class and later set its delegate instance by calling `ice_delegate`:

Java Compat

```
_NodeTie servant = new _NodeTie();      // Create tie
// ...
NodeI fred = new NodeI("Fred");         // Create implementation
// ...
servant.ice_delegate(fred);             // Set delegate
```

When using tie classes, it is important to remember that the tie instance is the servant, not your delegate. Furthermore, you must not use a tie instance to incarnate an Ice object until the tie has a delegate. Once you have set the delegate, you must not change it for the lifetime of the tie; otherwise, undefined behavior results.

You should use the tie approach only when necessary, that is, if you need to extend some base class in order to implement your servants: using the tie approach is more costly in terms of memory because each Ice object is incarnated by two Java objects (the tie and the delegate) instead of just one. In addition, call dispatch for ties is marginally slower than for ordinary servants because the tie forwards each operation to the delegate, that is, each operation invocation requires two function calls instead of one.

Also note that, unless you arrange for it, there is no way to get from the delegate back to the tie. If you need to navigate back to the tie from the delegate, you can store a reference to the tie in a member of the delegate. (The reference can, for example, be initialized by the constructor of the delegate.)

See Also

- Server-Side Java Compat Mapping for Interfaces
- Parameter Passing in Java Compat
- Raising Exceptions in Java Compat
- Object Incarnation in Java Compat

Object Incarnation in Java Compat

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a Java Servant

Instantiating a servant means to allocate an instance:

Java Compat

```
Node servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance and assigns its address to a reference of type `Node`. This works because `NodeI` is derived from `Node`, so a `Node` reference can refer to an instance of type `NodeI`. However, if we want to invoke a member function of the `NodeI` class at this point, we must use a `NodeI` reference:

Java Compat

```
NodeI servant = new NodeI("Fred");
```

Whether you use a `Node` or a `NodeI` reference depends purely on whether you want to invoke a member function of the `NodeI` class: if not, a `Node` reference works just as well as a `NodeI` reference.

Creating an Identity in Java

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

Java Compat

```
Identity id = new Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

Activating a Java Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

Java Compat

```
_adapter.add(servant, id);
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

UUIDs as Identities in Java

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. Java provides a helper function that we can use to create such identities:

Java Compat

```
public class Example
{
    public static void main(String[] args)
    {
        System.out.println(java.util.UUID.randomUUID().toString());
    }
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `randomUUID` creates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

Java

```
_adapter.addWithUUID(new NodeI("Fred"));
```

Creating Proxies in Java

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Hello World Application. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in Java

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write the following:

Java Compat

```
NodePrx proxy = NodePrxHelper.uncheckedCast(_adapter.addWithUUID(new
NodeI("Fred")));
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `ObjectPrx`.

Direct Proxy Creation in Java

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

Java Compat

```
Identity id = new Identity();
id.name = java.util.UUID.randomUUID().toString();
ObjectPrx o = _adapter.createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `randomUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Hello World Application
- Server-Side Java Compat Mapping for Interfaces
- Object Adapter States
- Servant Locators
- Object Life Cycle

Asynchronous Method Dispatch (AMD) in Java Compat

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool. If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server can provide its results to the Ice run time for delivery to the client.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

On this page:

Enabling AMD with Metadata in Java

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

Slice

```
["amd"] interface I
{
    bool isValid();
    float computeRate();
}

interface J
{
    ["amd"] void startProcess();
    int endProcess();
}
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface) minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

AMD Mapping in Java

The mapping emits the following code for each AMD operation:

1. Callback interface
2. Dispatch method

Callback interface for AMD

A callback interface is used by the implementation to notify the Ice run time about the completion of an operation. The name of this interface is formed using the pattern `AMD_class_op`. For example, an operation named `foo` defined in interface `I` results in an interface named `AMD_I_foo`. The interface is generated in the same scope as the interface or class containing the operation. Two methods are provided:

Java Compat

```
public void ice_response(<params>);
```

The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-`void` return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's out parameters follow the return value, in the order of declaration.

Java Compat

```
public void ice_exception(java.lang.Exception ex);
```

The `ice_exception` method allows the server to raise an exception. With respect to exceptions, there is less compile-time type safety in an AMD implementation because there is no `throws` clause on the dispatch method and any exception type could conceivably be passed to `ice_exception`. However, the Ice run time validates the exception value using the same semantics as for synchronous dispatch.

Neither `ice_response` nor `ice_exception` throw any exceptions to the caller.

Dispatch method for AMD

The dispatch method, whose name has the suffix `_async`, has a `void` return type. The first parameter is a reference to an instance of the callback interface described above. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

Slice

```
interface I
{
    ["amd"] int foo(short s, out long l);
}
```

The callback interface generated for operation `foo` is shown below:

Java Compat

```
public interface AMD_I_foo
{
    void ice_response(int __ret, long l);
    void ice_exception(java.lang.Exception ex);
}
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

Java Compat

```
void foo_async(AMD_I_foo __cb, short s);
```

AMD Exceptions in Java

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).

> These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the future or callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

AMD Example in Java

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

Slice

```
module Demo
{
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {}

    interface Model
    {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    }
}
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo._ModelDisp` and supplies a definition for the `interpolate_async` method that creates a `Job` to hold the callback object and arguments, and adds the `Job` to a queue. The method is synchronized to guard access to the queue:

Java Compat

```
public final class ModelI extends Demo._ModelDisp
{
    synchronized public void interpolate_async(
        Demo.AMD_Model_interpolate cb,
        float[][] data,
        float factor,
        Ice.Current current)
            throws RangeError
    {
        _jobs.add(new Job(cb, data, factor));
    }

    java.util.LinkedList<Job> _jobs = new java.util.LinkedList<Job>();
}
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

Java Compat

```
class Job
{
    Job(Demo.AMD_Model_interpolate cb,
        float[][] grid,
        float factor)
    {
        _cb = cb;
        _grid = grid;
        _factor = factor;
    }

    void execute()
    {
        if(!interpolateGrid())
        {
            _cb.ice_exception(new Demo.RangeError());
            return;
        }
        _cb.ice_response(_grid);
    }

    private boolean interpolateGrid()
    {
        // ...
    }

    private Demo.AMD_Model_interpolate _cb;
    private float[][] _grid;
    private float _factor;
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

See Also

- User Exceptions
- Run-Time Exceptions
- Asynchronous Method Invocation (AMI) in Java Compat
- The Ice Threading Model

Example of a File System Server in Java Compat

This page presents the source code for a Java server that implements our file system and communicates with the client we wrote earlier. The code is fully functional, apart from the required interlocking for threads.

Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

On this page:

Implementing a File System Server in Java

We have now seen enough of the server-side Java mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of three source files:

- `Server.java`
  This file contains the server main program.
- `Filesystem/DirectoryI.java`
  This file contains the implementation for the `Directory` servants.
- `Filesystem/FileI.java`
  This file contains the implementation for the `File` servants.

Server Main Program in Java

Our server main program, in the file `Server.java`, consists of two static methods, `main` and `run`. `main` creates and destroys an Ice communicator, and `run` uses this communicator instantiate our file system objects:

Java Compat

```
import Filesystem.*;


public class Server
{
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.StringSeqHolder argsHolder = new Ice.StringSeqHolder(args);

        //
        // Try with resources block - communicator is automatically
destroyed
        // at the end of this try block
        //
        try(final Ice.Communicator communicator =
```

```
Ice.Util.initialize(argsHolder))
        {
            //
            // Install shutdown hook to (also) destroy communicator
during JVM shutdown.
            // This ensures the communicator gets destroyed when the
user interrupts the application with Ctrl-C.
            //
            Runtime.getRuntime().addShutdownHook(new Thread()
                {
                    @Override
                    public void run()
                    {
                        communicator.destroy();
                        System.err.println("terminating");
                    }
                });

            status = run(communicator, argsHolder.value);
        }

        System.exit(status);
    }

    private static int
    run(Ice.Communicator communicator, String[] args)
    {
        //
        // Create an object adapter.
        //
        Ice.ObjectAdapter adapter =

communicator.createObjectAdapterWithEndpoints("SimpleFilesystem",
"default -h localhost -p 10000");

        //
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI root = new DirectoryI(communicator, "/", null);
        root.activate(adapter);

        //
        // Create a file called "README" in the root directory
        //
        FileI file = new FileI(communicator, "README", root);
        String[] text;
        text = new String[]{ "This file system contains a collection of
poetry." };
        try
        {
```

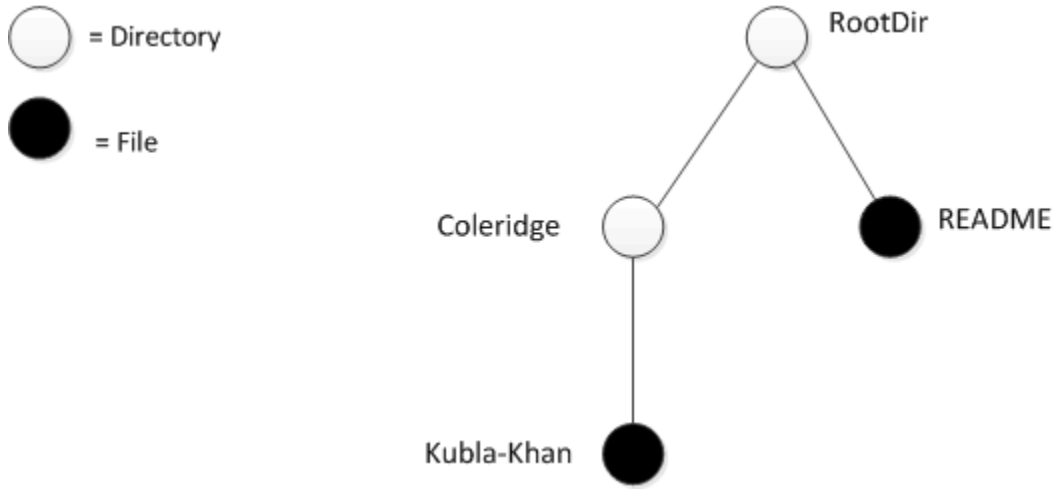
```
            file.write(text, null);
        }
        catch(GenericError e)
        {
            System.err.println(e.reason);
        }
        file.activate(adapter);

        //
        // Create a directory called "Coleridge" in the root directory
        //
        DirectoryI coleridge = new DirectoryI(communicator, "Coleridge",
root);
        coleridge.activate(adapter);

        //
        // Create a file called "Kubla_Khan" in the Coleridge directory
        //
        file = new FileI(communicator, "Kubla_Khan", coleridge);
        text = new String[]{ "In Xanadu did Kubla Khan",
                             "A stately pleasure-dome decree:",
                             "Where Alph, the sacred river, ran",
                             "Through caverns measureless to man",
                             "Down to a sunless sea." };
        try
        {
            file.write(text, null);
        }
        catch(GenericError e)
        {
            System.err.println(e.reason);
        }
        file.activate(adapter);

        //
        // All objects are created, allow client requests now
        //
        adapter.activate();

        //
        // Wait until we are done
        //
        communicator.waitForShutdown();

        return 0;
```

```
        }
    }
```

The code imports the contents of the `Filesystem` package. This avoids having to continuously use fully-qualified identifiers with a `Filesystem.` prefix.

The next part of the source code is the definition of the `Server` class. Much of this code is boiler plate: we create a communicator, then an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

Java Compat

```
DirectoryI root = new DirectoryI("/", null);
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in the above illustration:

Java Compat

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI("/", null);

// Create a file "README" in the root directory
//
File file = new FileI("README", root);
String[] text;
text = new String[]
{
    "This file system contains a collection of poetry."
};
try
{
    file.write(text, null);
}
catch(GenericError e)
{
    System.err.println(e.reason);
}

// Create a directory "Coleridge" in the root directory
//
DirectoryI coleridge = new DirectoryI("Coleridge", root);

// Create a file "Kubla_Khan" in the Coleridge directory
//
file = new FileI("Kubla_Khan", coleridge);
text = new String[]{ "In Xanadu did Kubla Khan",
                     "A stately pleasure-dome decree:",
                     "Where Alph, the sacred river, ran",
                     "Through caverns measureless to man",
                     "Down to a sunless sea." };
try
{
    file.write(text, null);
}
catch(GenericError e)
{
    System.err.println(e.reason);
}
```

We first create the root directory and a file `README` within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

Java Compat

```
        String[] text;
        text = new String[]
        {
            "This file system contains a collection of poetry."
        };
        try
        {
            file.write(text, null);
        }
        catch(GenericError e)
        {
            System.err.println(e.reason);
        }
```

Recall that Slice sequences by default map to Java arrays. The Slice type `Lines` is simply an array of strings; we add a line of text to our `README` file by initializing the `text` array to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by writing:

Java Compat

```
            file.write(text, null);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Java function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the illustration listed above.

`FileI` Servant Class in Java

Our `FileI` servant class has the following basic structure:

Java Compat

```
public class FileI extends _FileDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private String[] _lines;
}
```

The class has a number of data members:

- `_adapter`
  This static member stores a reference to the single object adapter we use in our server.
- `_name`
  This member stores the name of the file incarnated by the servant.
- `_parent`
  This member stores the reference to the servant for the file's parent directory.
- `_lines`
  This member holds the contents of the file.

The `_name` and `_parent` data members are initialized by the constructor:

Java Compat

```
public
FileI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    assert(_parent != null);

    // Create an identity
    //
    Ice.Identity myID = new Ice.Identity();
    myID.name = java.util.UUID.randomUUID().toString();

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and
    // add it as a child to the parent
    //
    NodePrx thisNode =
NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
    _parent.addChild(thisNode);
}
```

After initializing the `_name` and `_parent` members, the code verifies that the reference to the parent is not null because every file must have a parent directory. The constructor then generates an identity for the file by calling `java.util.UUID.randomUUID` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function in `DirectoryI` Methods below.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

Java Compat

```
// Slice Node::name() operation

public String
name(Ice.Current current)
{
    return _name;
}

// Slice File::read() operation

public String[]
read(Ice.Current current)
{
    return _lines;
}

// Slice File::write() operation

public void
write(String[] text, Ice.Current current)
    throws GenericError
{
    _lines = text;
}
```

The `name` method is inherited from the generated `Node` interface. It returns the value of the `_name` member.

The `read` and `write` methods are inherited from the generated `File` interface and return and set the `_lines` member.

`DirectoryI` Servant Class in Java

The `DirectoryI` class has the following basic structure:

Java Compat

```
package Filesystem;

public final class DirectoryI extends _DirectoryDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private java.util.ArrayList<NodePrx> _contents = new
java.util.ArrayList<NodePrx>();
}
```

`DirectoryI` Data Members

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds a null reference.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

Java Compat

```
public
DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    // Create an identity. The parent has the
    // fixed identity "RootDir"
    //
    Ice.Identity myID = new Ice.Identity();
    myID.name = _parent != null ? java.util.UUID.randomUUID().toString()
: "RootDir";

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and add it as a
    // child to the parent
    //
    NodePrx thisNode =
NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
    if(_parent != null)
    {
        _parent.addChild(thisNode);
    }
}
```

`DirectoryI` Constructor

The constructor creates an identity for the new directory by calling `java.util.UUID.randomUUID`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a reference to itself and passes it to the `addChild` helper function.

`DirectoryI` Methods

`addChild` adds the passed reference to the `_contents` list:

Java Compat

```
void addChild(NodePrx child)
{
    _contents.add(child);
}
```

The remainder of the operations, `name` and `list`, are equally trivial:

Java Compat

```
public String
name(Ice.Current current)
{
    return _name;
}

// Slice Directory::list() operation

public NodePrx[]
list(Ice.Current current)
{
    NodePrx[] result = new NodePrx[_contents.size()];
    _contents.toArray(result);
    return result;
}
```

Note that the `_contents` member is of type `java.util.ArrayList<NodePrx>`, which is convenient for the implementation of the `addChild` method. However, this requires us to convert the list into a Java array in order to return it from the `list` operation.

See Also

- Slice for a Simple File System
- Example of a File System Client in Java Compat
- Java Compat Mapping for Sequences
- The Ice Threading Model

Slice-to-Java Compat Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to Java Compat is identical to the mapping for these constructs without the `local` qualifier. The generated Java code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

The rest of this section describes the mapping of the remaining local types to Java Compat:

- Java Compat Mapping for Local Interfaces
- Java Compat Mapping for Local Classes
- Java Compat Mapping for Local Exceptions
- Java Compat Mapping for Operations on Local Types
- Java Compat Mapping for Data Members in Local Types

Java Compat Mapping for Local Interfaces

On this page:

Mapped Java Class

A Slice local interface is mapped to a Java interface with the same name, for example:

Slice

```
module Ice
{
    local interface Communicator
    {
        ...
    }
}
```

is mapped to the Java interface `Communicator`:

Java

```
package Ice;

public interface Communicator
{
    ...
}
```

LocalObject in Java

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to `java.lang.Object` in Java.

Mapping for Local Interface Inheritance in Java

Inheritance of local Slice interfaces is mapped to interface inheritance in Java. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

Java

```
package M;

public interface A {}
public interface B extends A {}
public interface C extends A {}
public interface D extends B, C {}
```

Java Compat Mapping for Local Classes

On this page:

Mapped Java Class

A local Slice class is mapped to a concrete or abstract Java class with the same name. For example:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ...
    }
}
```

is mapped to the Java class `ConnectionInfo`:

Java

```
package Ice;

public class ConnectionInfo implements java.lang.Cloneable
{
    ...
}
```

All mapped Java classes implement `java.lang.Cloneable.`

LocalObject in Java

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to `java.lang.Object` in Java.

Mapping for Local Interface Inheritance in Java

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` for classes is mapped to `extends` in Java, and `implements` is mapped to `implements`. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

Java

```
package M;

public interface A {}
public interface B {}

public class C implements A, B, java.lang.Cloneable {}
public class D extends C {}
```

Java Compat Mapping for Local Exceptions

On this page:

Mapped Java Class

A local Slice exception is mapped to a Java class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the Java class `InitializationException`:

Java

```
package Ice;

public class InitializationException extends LocalException
{
    ...
}
```

Base Class for Local Exceptions in Java

All mapped Java classes for local exceptions extend the abstract class `LocalException`:

Java

```
package Ice;

public abstract class LocalException extends Exception
{
    public LocalException() {}
    public LocalException(Throwable cause) { ... }
}
```

Mapping for Local Exception Inheritance in Java

A local Slice exception can extend another Slice exception, which is mapped to `extends` in Java. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

Java

```
package M;
public class ErrorBase extends com.zeroc.Ice.LocalException { ... }
public class ResourceError extends ErrorBase { ... }
```

Java Compat Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to a Java method with the same name. The mapping of operation parameters to Java is identical to the Client-Side Mapping for these parameters, in particular, out parameters are mapped to Java `Holder` parameters.

Unlike the Client-Side mapping, there is no mapped method with a trailing Context parameter.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

Java

```
package M;

public final class LSeqHolder extends Ice.Holder<L[]> { ... }
public interface L
{
    String op(int n, String s, java.lang.Object any, Ice.IntHolder m,
Ice.StringHolder t, LSeqHolder newLSeq);
}
```

Java Compat Mapping for Data Members in Local Types

Data members on local Slice types (classes, exceptions and structs) are mapped to Java just like the data members of the corresponding non local Slice construct.

A local Slice type can have a data member of type local interface or class: it is mapped a Java data member with the mapped Java interface or class as its type.

The Util Class in Java Compat

Ice for Java includes a number of utility APIs in the `Ice.Util` class.

This page summarizes the contents of these APIs for your reference.

The `Util` Class

Communicator Initialization Methods

`Util` provides a number of overloaded `initialize` methods that create a communicator.

Identity Conversion

`Util` contains two methods for converting object identities of type `Identity` to and from strings.

Per-Process Logger Methods

`Util` provides methods for getting and setting the per-process logger.

Property Creation Methods

`Util` provides a number of overloaded `createProperties` methods that create property sets.

Proxy Comparison Methods

Two methods, `proxyIdentityCompare` and `proxyIdentityAndFacetCompare`, allow you to compare object identities that are stored in proxies (either ignoring the facet or taking the facet into account).

Version Information

The `stringVersion` and `intVersion` methods return the version of the Ice run time:

Java

```
public static String stringVersion();
public static int intVersion();
```

The `stringVersion` method returns the Ice version in the form <*major*>.<*minor*>.<*patch*>, for example, `3.7.1`. For beta releases, the version is <*major*>.<*minor*>b, for example, `3.7b`.

The `intVersion` method returns the Ice version in the form *AABBCC*, where *AA* is the major version number, *BB* is the minor version number, and *CC* is patch level, for example, `30701` for version 3.7.1. For beta releases, the patch level is set to 51 so, for example, for version 3.7b, the value is `30751`.

See Also

- Java Compat Mapping for Interfaces
- Command-Line Parsing and Initialization
- Setting Properties
- Object Identity
- Java Streaming Interfaces

Custom Class Loaders in Java Compat

Certain features of the Ice for Java run-time necessitate dynamic class loading. Applications with special requirements can supply a custom class loader for Ice to use in the following situations:

- Unmarshaling user exceptions and instances of concrete Slice classes
- Loading Ice plug-ins
- Loading IceSSL certificate verifiers and password callbacks

If an application does not supply a class loader (or if the application-supplied class loader fails to locate a class), the Ice run time attempts to load the class using class loaders in the following order:

- current thread's class loader
- default class loader (that is, by calling `Class.forName`)
- system class loader

Note that an application must install object factories for any abstract Slice classes it might receive, regardless of whether the application also installs a custom class loader.

To install a custom class loader, set the `classLoader` member of `Ice.InitializationData` prior to creating a communicator:

Java Compat

```
Ice.InitializationData initData = new Ice.InitializationData();
initData.classLoader = new MyClassLoader();
Ice.Communicator communicator = Ice.Util.initialize(args, initData);
```

See Also

- Java Compat Mapping for Exceptions
- Java Compat Mapping for Classes
- Plug-in Facility
- IceSSL
- Communicator Initialization

Handling Interrupts in Java Compat

Java applications can safely interrupt a thread that calls a blocking Ice API.

> You must enable the `Ice.ThreadInterruptSafe` property to use interrupts. Enabling interrupts causes Ice to disable message buffer caching, which may incur a slight performance penalty.

With interrupt support enabled, Ice guarantees that every call into the run time that could potentially block indefinitely is an *interruption point*. On entry, an interruption point raises the unchecked exception `OperationInterruptedException` immediately if the current thread's interrupted flag is true. If the application interrupts the run time after the interruption point has begun its processing, the interruption point will either raise `OperationInterruptedException` or return control to the application with the thread's interrupted flag set. Under no circumstances will Ice silently discard an interrupt.

For example, the following code shows how to interrupt a thread that's blocked while making a synchronous proxy invocation:

Java Compat

```
Thread proxyThread = Thread.currentThread();
AccountPrx account = // get proxy...

try
{
    String name = account.getName(); // Blocks calling thread
    ...
}
catch(Ice.OperationInterruptedException ex)
{
   // The invocation was interrupted.
}

// At this point either the invocation was interrupted with an
// OperationInterruptedException or proxyThread.isInterrupted() is true.

// In another thread either just prior to or during the getName()
invocation.
proxyThread.interrupt();
```

Here is a similar example that uses the asynchronous proxy API instead:

Java Compat

```
Thread proxyThread = Thread.currentThread();
AccountPrx account = // get proxy...

Ice.AsyncResult r = account.begin_getName(); // Never blocks

try
{
    // do more work
    String name = account.end_getName(r); // May block calling thread
    ...
}
catch(Ice.OperationInterruptedException ex)
{
    // The invocation was interrupted.
}

// At this point either the invocation was interrupted with an
// OperationInterruptedException or proxyThread.isInterrupted() is true.

// In another thread either just prior to or during the getName()
invocation.
proxyThread.interrupt();
```

Synchronous proxy invocations are interruption points. For an asynchronous proxy invocation, the `begin_` method never blocks and therefore it is not an interruption point, however the `end_` method is an interruption point.

Additionally, all of the methods listed in Blocking API Calls are also interruption points. In the specific case of `Communicator.destroy()`, we recommend calling `destroy` in a loop as shown below:

Java Compat

```
while(true)
{
    try
    {
        communicator.destroy();
        break;
    }
    catch(Ice.OperationInterruptedException ex)
    {
        continue;
    }
}
```

Interrupting a call to `destroy` could leave the Ice run time in a partially-destroyed state; calling `destroy` again as we've done here ensures Ice can finish reclaiming the communicator's resources.

See Also

JavaScript Mapping

Supported Features of the JavaScript Mapping

Ice for JavaScript does not support the entire set of Ice features that are found in the C++, Java, and C# language mappings, primarily due to platform and API limitations. Furthermore, there are some differences between using Ice for JavaScript in a browser and in Node.js.

The table below provides more details on the Ice for JavaScript feature set:

Feature	Browser	Node.js	Notes
Synchronous invocations			Blocking RPCs are not compatible with JavaScript's execution model.
Asynchronous invocations	✅	✅	Also see Asynchronous APIs in JavaScript
Synchronous dispatch	✅	✅	
Asynchronous dispatch	✅	✅	
Outgoing TCP connections		✅	
Incoming TCP connections			Applications must use bidirectional connections to receive callbacks.
Outgoing WS connections	✅		WebSocket connections require the server to configure a Websocket endpoint.
Outgoing WSS connections	✅		Secure WebSocket (WSS) connections require the server to configure IceSSL and a Websocket endpoint.
Outgoing SSL connections			Browser applications can use WSS.
Datagrams			
DNS queries	⚠	⚠	The lack of support for DNS queries affects fault tolerance.
File logging			
Property files			
Flow control API			"Sent" callbacks are not supported.
Thread pools			Threads are not supported in JavaScript.
`PerThread` implicit context			Threads are not supported in JavaScript.
Protocol compression			Ice uses the bzip2 algorithm, which is not natively supported in JavaScript.
Communicator plug-ins			
`Dispatcher` API			
`DispatchInterceptor` API			
Collocated invocations			
Streaming API			
Admin facility			

DNS limitations

In most language mappings, the Ice run time performs a DNS query to resolve an endpoint's host name into one or more IP addresses. Specifying a multi-homed host name in an endpoint provides the client with a simple form of fault tolerance because Ice has multiple addresses to use during connection establishment.

This form of fault tolerance is not available when using Ice for JavaScript because the DNS API is not supported.

Topics

Asynchronous APIs in JavaScript

Given JavaScript's lack of support for threads, Ice for JavaScript uses asynchronous semantics in every situation that has the potential to block, including both local and non-local invocations. Synchronous proxy invocations are not supported.

Here is an example of a simple proxy invocation:

JavaScript

```
const proxy = HelloPrx.uncheckedCast(...);
const promise = proxy.sayHello();
promise.then(
    () => {
        // handle success...
    },
    ex => {
        // handle failure...
    }
);
```

The API design is similar to that of other asynchronous Ice language mappings in that the return value of a proxy invocation is an instance of `Ice.AsyncResult`, through which the program configures callbacks to handle the eventual success or failure of the invocation. The JavaScript implementation of `Ice.AsyncResult` derives from the standard JavaScript Promise.

We can simplify the example above using the `await` keyword:

JavaScript

```
(async function()
{
    const proxy = HelloPrx.uncheckedCast(...);
    try
    {
        await proxy.sayHello();
        // handle success...
    }
    catch(ex)
    {
        // handle failure...
    }
}());
```

Note that the `await` keyword can only be used in a function marked with the `async` keyword.

Certain operations of local Ice run-time objects can also block, either because their implementations make remote invocations, or because their purpose is to block until some condition is satisfied. See Blocking API Calls for a list of these operations. Like proxy invocations, these operations return the standard JavaScript Promise. The example below shows how to call `ice_getConnection` on a proxy:

JavaScript

```
const communicator = ...;
const proxy = communicator.stringToProxy("...");
proxy.ice_getConnection().then(
    connection => {
        // got connection...
    },
    ex => {
        // connection failed...
    }
);
```

Client-Side Slice-to-JavaScript Mapping

In this section, we present the client-side Slice-to-JavaScript mapping. The client-side Slice-to-JavaScript mapping defines how Slice data types are translated to JavaScript types, and how clients invoke operations, pass parameters, and handle errors. Much of the JavaScript mapping is intuitive. For example, Slice sequences map to JavaScript arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in JavaScript.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

In order to use the JavaScript mapping, you should need no more than the Slice definition of your application and knowledge of the JavaScript mapping rules. In particular, looking through the generated code in order to discern how to use the JavaScript mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

The `Ice` Scope
All of the APIs for the Ice run time are nested in the `Ice` scope, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` scope are generated from Slice definitions; other parts of the `Ice` scope provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` scope throughout the remainder of the manual.

Topics

JavaScript Mapping for Identifiers

A Slice identifier maps to an identical JavaScript identifier. For example, the Slice identifier `Clock` becomes the JavaScript identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a JavaScript keyword or is an identifier reserved by the Ice run time (such as `checkedCast`), the corresponding JavaScript identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.

> You should try to avoid such identifiers as much as possible.

A single Slice identifier often results in several JavaScript identifiers. For example, for a Slice interface named `Foo`, the generated JavaScript code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_while` and `whilePrx` (*not* `_whilePrx`), that is, the underscore prefix is applied only to those generated identifiers that actually require it.

See Also

- Lexical Rules
- JavaScript Mapping for Modules
- JavaScript Mapping for Built-In Types
- JavaScript Mapping for Enumerations
- JavaScript Mapping for Structures
- JavaScript Mapping for Sequences
- JavaScript Mapping for Dictionaries
- JavaScript Mapping for Constants
- JavaScript Mapping for Exceptions

JavaScript Mapping for Modules

On this page:

Default Mapping for Modules

The default mapping for a Slice module maps to a JavaScript object with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions. For example:

Slice

```
module M1
{
    // Definitions for M1 here...
    module M2
    {
        // Definitions for M2 here...
    }
}

// ...

module M1    // Reopen M1
{
    // More definitions for M1 here...
}
```

This definition maps to the corresponding JavaScript definitions:

JavaScript

```
(function(module, require, exports)
{
    var Ice = ...
    var _ModuleRegistry = Ice._ModuleRegistry;
    var M1 = _ModuleRegistry.module("M1");

    // Definitions for M1 here...

    M1.M2 = _ModuleRegistry.module("M1.M2");

    // Definitions for M2 here...

    // More definitions for M1 here...

    exports.M1 = M1;
}(...));
```

The generated code exports the top-level modules. For browsers it adds symbols to the global window object and for NodeJS it uses the native exports object.

When developing for NodeJS, you include the various Ice components with `require` statements as shown below:

JavaScript

```
var Ice = require("ice").Ice;
var IceGrid = require("ice").IceGrid;
var IceStorm = require("ice").IceStorm;
...
```

Importing the generated code for your own Slice definitions works the same way:

JavaScript

```
var M1 = require("M1Defs").M1;
```

This example assumes the generated Slice definitions are in the file `M1Defs.js`.

In a browser application, the necessary JavaScript files are usually loaded via HTML `script` tags:

```
<script type="text/javascript" src="Ice.js"></script>
<script type="text/javascript" src="M1Defs.js"></script>
<script type="text/javascript">
    // You can now access Ice and M1 as global objects
</script>
```

The file `Ice.js` shown above only provides definitions for the Ice run time; you would need to explicitly include the corresponding files for

any other Ice components that your application needs, such as `IceGrid.js`.

Alternate Mapping for Modules

Ice 3.7 introduces support for an alternative mapping for Slice modules. The alternate mapping differs from the standard mapping in the way that modules are imported and exported. With this new mapping the import and export are done using the standard JavaScript import and export statements.

Like with the default mapping, each Slice module maps to a JavaScript object with the same name, and only the top-level modules are exported. Using our previous example we need to add the `[["js:es6-module"]]` global metadata to enable the new module mapping:

Slice

```
[["js:es6-module"]]
module M1
{
    // Definitions for M1 here...
    module M2
    {
        // Definitions for M2 here...
    }
}

// ...

module M1    // Reopen M1
{
    // More definitions for M1 here...
}
```

This definition maps to the corresponding JavaScript definitions:

JavaScript

```
import { Ice } from "ice";
const _ModuleRegistry = Ice._ModuleRegistry;
//...
let M1 = _ModuleRegistry.module("M1");
//...
M1.M2 = _ModuleRegistry.module("M1.M2");
// ...
export { M1 };
```

Using ES6 modules in the browser

When using the es6 module mapping in a browser you will typically use it in conjuntion with a JavaScript module bundler like WebPack or Rollup. You will need to define a global "ice" object see sample configurations bellow:

WebPack Rollup

```
// webpack.config.js
const path = require('path');

module.exports = {
  entry: './main.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  externals: {
      "ice": "window"
  }
};
```

See https://webpack.js.org/configuration/externals/

```
// rollup.config.js
export default {
    input: 'main.js',
    output: {
        file: 'bundle.js',
        format: 'iife',
        globals: {
            ice: 'window'
        }
    },
    external: ['ice'],
};
```

See https://rollupjs.org/

See Also

- Modules
- Using the Slice Compilers
- JavaScript Mapping for Identifiers
- JavaScript Mapping for Built-In Types
- JavaScript Mapping for Enumerations
- JavaScript Mapping for Structures
- JavaScript Mapping for Sequences
- JavaScript Mapping for Dictionaries
- JavaScript Mapping for Constants
- JavaScript Mapping for Exceptions

JavaScript Mapping for Built-In Types

On this page:

Mapping of Slice Built-In Types to JavaScript Types

The Slice built-in types are mapped to JavaScript types as follows:

Slice	JavaScript
bool	`Boolean`
byte	`Number`
short	`Number`
int	`Number`
long	`Ice.Long`
float	`Number`
double	`Number`
string	`String`

Mapping of Slice built-in types to JavaScript.

JavaScript Mapping for Long Integers

JavaScript does not provide a type that is capable of fully representing a Slice `long` value (a 64-bit integer), therefore Ice provides the `Ice.Long` type. An instance of `Ice.Long` encapsulates the high and low order 32-bit words (in little endian format) representing the 64-bit `long` value and provides access to these values via the `high` and `low` properties. Instances also provide the `toNumber` method, which returns a `Number` if the long value can be correctly represented by the `Number` type's integer range, otherwise it returns `Number.POSITIVE_INFINITY` or `Number.NEGATIVE_INFINITY` for positive and negative values, respectively.

`Ice.Long` objects have limited functionality because their primary purpose is to allow a `long` value to be re-transmitted, but instances do support the methods `hashCode`, `equals`, and `toString`.

See Also

- Basic Types
- JavaScript Mapping for Identifiers
- JavaScript Mapping for Modules
- JavaScript Mapping for Enumerations
- JavaScript Mapping for Structures
- JavaScript Mapping for Sequences
- JavaScript Mapping for Dictionaries
- JavaScript Mapping for Constants
- JavaScript Mapping for Exceptions

JavaScript Mapping for Enumerations

JavaScript does not have an enumerated type, so a Slice enumeration is emulated using JavaScript objects where each enumerator is an instance of the same type. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The generated JavaScript code is equivalent to the following:

JavaScript

```
class Fruit { ... }
Fruit.Apple = new Fruit("Apple", 0);
Fruit.Pear = new Fruit("Pear", 1);
Fruit.Orange = new Fruit("Orange", 2);
```

Each enumerator defines `name` and `value` properties that supply the enumerator's name and ordinal value, respectively. Enumerators also define `hashCode`, `equals` and `toString` methods, and the enumerated type itself defines a `valueOf` method that converts ordinal values into their corresponding enumerators.

Suppose we modify the Slice definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The generated code changes accordingly:

JavaScript

```
class Fruit = { ... };
Fruit.Apple = new Fruit("Apple", 0);
Fruit.Pear = new Fruit("Pear", 3);
Fruit.Orange = new Fruit("Orange", 4);
```

Given the above definitions, we can use enumerated values as follows:

JavaScript

```
const f1 = Fruit.Apple;
const f2 = Fruit.Orange;

if(f1 === Fruit.Apple)   // Compare with constant
{
    // ...
}

if(f1 === f2)            // Compare two enums
{
    // ...
}

switch(f2)               // Switch on enum
{
    case Fruit.Apple:
        // ...
        break;
    case Fruit.Pear:
        // ...
        break;
    case Fruit.Orange:
        // ...
        break;
}

const f = Fruit.valueOf(3); // Convert an ordinal value to its
enumerator, or undefined if no match
console.log(f.name + " = " + f.value); // Outputs "Pear = 3"
```

Note that the generated type may contain other members, which we have not shown. These members are internal to the Ice run time and you must not use them in your application code (because they may change from release to release).

See Also

- Enumerations
- JavaScript Mapping for Structures
- JavaScript Mapping for Sequences
- JavaScript Mapping for Dictionaries

JavaScript Mapping for Structures

On this page:

Basic JavaScript Mapping for Structures

A Slice structure maps to a JavaScript type with the same name. For each Slice data member, the JavaScript instance contains a corresponding property. As an example, here is our Employee structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The JavaScript mapping for this structure is equivalent to the following code:

JavaScript

```
class Employee
{
    function(number = new Ice.Long(0, 0), firstName = "", lastName = "")
    {
        this.number = number;
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

For each data member in the Slice definition, the JavaScript constructor assigns a value to its corresponding property. Instances define an `equals` method for comparison purposes and a `clone` method to create a shallow copy.

If the structure qualifies for use as a dictionary key, instances also define a `hashCode` function as required by the `Ice.HashMap` type.

JavaScript Constructors for Structures

The generated constructor has one parameter for each data member. This allows you to construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members). The constructor assigns a default value to each property appropriate for the type of its corresponding member:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value

Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The constructor initializes each of these data members to its declared value instead.

See Also

- Structures
- JavaScript Mapping for Enumerations
- JavaScript Mapping for Sequences
- JavaScript Mapping for Dictionaries

JavaScript Mapping for Sequences

On this page:

Basic JavaScript Mapping for Sequences

A Slice sequence maps to a native JavaScript array. This means that the Slice-to-JavaScript compiler does not generate a separate named type for a Slice sequence. It also means that you can take advantage of all the inherent functionality offered by JavaScript's array type. For example:

Slice

```
sequence<Fruit> FruitPlatter;
```

We can use the `FruitPlatter` sequence as shown below:

JavaScript

```
const platter = [Fruit.Apple];
platter.push(Fruit.Pear);
```

JavaScript Mapping for Byte Sequences

As an optimization, occurrences of `sequence<byte>` map to a Uint8Array type, which is more efficient than native arrays for working with binary data.

See Also

- Sequences
- JavaScript Mapping for Enumerations
- JavaScript Mapping for Structures
- JavaScript Mapping for Dictionaries

JavaScript Mapping for Dictionaries

Here is the definition of our EmployeeMap once more:

Slice

```
dictionary<int, Employee> EmployeeMap;
```

In the JavaScript mapping, dictionaries using a JavaScript built-in type as the key type are mapped to the JavaScript Map type. This is true for all Slice built-in types except `long`:

JavaScript

```
const em = new Map();

const e = new Employee();
e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.set(e.number, e);
```

For cases where the key type does not correspond with a JavaScript built-in type, the dictionary is mapped to HashMap. This is true for Slice dictionaries where the key type is `long` or a Slice structure that qualifies as a legal dictionary key:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

In these cases an extra constructor is generated that initializes the HashMap with the desired comparison operator.

JavaScript

```
const em = new EmployeeMap();

const e = new Employee();
e.number = new Ice.Long(31);
e.firstName = "James";
e.lastName = "Gosling";

em.set(e.number, e);
```

HashMap supports the same API as the standard JavaScript Map object. It provides the following additional properties and functions:

- `HashMap.prototype.constructor(keyComparator, valueComparator)`
  This version of the constructor accepts optional comparator functions that the map uses to compare keys and values for equality. If you instantiate a map directly using `new Ice.HashMap()` without specifying comparator functions, the default comparators use the === operator to compare keys and values. As an example, the following map compares its keys and values using `equals` meth ods:

JavaScript

```
function compareEquals(a, b)
{
    return a.equals(b);
}
const m = new Ice.HashMap(compareEquals, compareEquals);
```

The `valueComparator` function is only used when comparing two maps for equality.

> The type-specific constructor generated for a Slice dictionary supplies comparator functions appropriate for its key and value types.

- `HashMap.prototype.equals(other, valueComparator)`
  Returns true if this map compares equal to the given map, false otherwise. You can optionally supply a function for `valueComparator` that the map uses when comparing values; this function takes precedence over the comparator supplied to the map's constructor.
- `HashMap.prototype.clone()`
  Returns a shallow copy of the map.

Legal key types for `HashMap` include JavaScript's primitive types along with `null`, `NaN`, and any object that defines a `hashCode` method. The generated code for a Slice structure that qualifies as a legal dictionary key type includes a `hashCode` method. Suppose we define another dictionary type:

Slice

```
dictionary<Employee, string> EmployeeDeptMap;
```

The Slice compiler generates a constructor function equivalent to the following code:

JavaScript

```
class EmployeeDeptMap extends Ice.HashMap
{
    constructor(h)
    {
        const keyComparator = ...;
        const valueComparator = ...;
        super(h || keyComparator, valueComparator);
    }
}
```

Since the key is a user-defined structure type, the map requires a comparator that properly compares keys. Instantiating a map using `new EmployeeDeptMap` automatically sets the comparators, whereas calling `new Ice.HashMap` in this case would require you to supply your own comparators.

> Slice dictionaries that map to a HashMap must be instantiated using the generated constructor.

Notes

- Attempting to use the `map[key] = value` syntax to add an element to the map will not have the desired effect; you must use the `set` function instead.

See Also

- Dictionaries
- JavaScript Mapping for Enumerations
- JavaScript Mapping for Structures
- JavaScript Mapping for Sequences

JavaScript Mapping for Constants

Here are the sample constant definitions once more:

Slice

```
module Example
{
    const bool      AppendByDefault = true;
    const byte      LowerNibble = 0x0f;
    const string    Advice = "Don't Panic!";
    const short     TheAnswer = 42;
    const double    PI = 3.1416;

    enum Fruit { Apple, Pear, Orange }
    const Fruit     FavoriteFruit = Pear;
}
```

For each constant, the JavaScript mapping generates a read-only property of the same name in the enclosing scope as shown below:

JavaScript

```
Object.defineProperty(Example, 'AppendByDefault', {value: true});
Object.defineProperty(Example, 'LowerNibble', {value: 15});
Object.defineProperty(Example, 'Advice', {value: "Don't Panic!"});
Object.defineProperty(Example, 'TheAnswer', {value: 42});
Object.defineProperty(Example, 'PI', {value: 3.1416});
Object.defineProperty(Example, 'FavoriteFruit', {value: Fruit.Pear});
```

Slice string literals that contain non-ASCII characters or universal character names are mapped to JavaScript string literals with universal character names. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

JavaScript

```
Object.defineProperty(Example, 'Egg', {value: "\u0153uf"});
Object.defineProperty(Example 'Heart', {value: "c\u0153ur"});
Object.defineProperty(Example, 'Banana', {value: "\ud83c\udf4c"});
```

See Also

JavaScript Mapping for Exceptions

On this page:

JavaScript Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

JavaScript

```
class GenericError extends Ice.UserException
{
    constructor(reason = "", _cause = ""){ ... }
    ice_name()
    {
        return "M::GenericError";
    }
}

class BadTimeVal extends GenericError
{
    constructor(reason, _cause = ""){ ... }
    ice_name()
    {
        return "M::BadTimeVal";
    }
}

class BadZoneName extends GenericError
{
    constructor(reason, _cause = ""){ ... }
    ice_name()
    {
        return "M::BadZoneName";
    }
}
```

Each Slice exception maps to a JavaScript class with the same name. Each data member of an exception corresponds to a property in the JavaScript type.

The inheritance structure of the Slice exceptions is preserved for the generated types, so the prototypes for `BadTimeVal` and `BadZoneNam e` inherit from `GenericError`.

Each exception also defines the `ice_name` member function, which returns the name of the exception.

All user exceptions ultimately derive from the base type `Ice.UserException`. This allows you to handle any user exception generically as an `Ice.UserException`. Similarly, you can handle any Ice run-time exceptions as an `Ice.LocalException`, and you can handle any Ice exception as an `Ice.Exception`.

Note that the generated exception types may contain other properties and functions that are not shown. However, these elements are internal to the JavaScript mapping and are not meant to be used by application code.

JavaScript Constructors for User Exceptions

The generated constructor has one parameter for each data member. This allows you to construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members). For derived exceptions, the constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

The constructor assigns a default value to each property appropriate for the type of its corresponding member:

Data Member Type	Default Value

`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The constructor initializes each of these data members to its declared value instead.

The generated constructor accepts an optional trailing argument representing the entity that caused the exception. This value is typically an instance of `Error`, but that is not a requirement. The value is used to initialize the `ice_cause` property inherited from `Ice.Exception`.

JavaScript Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice.LocalException` (which, in turn, derives from `Ice.Exception`).

Recall the inheritance diagram for user and run-time exceptions. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice.Exception`
  This is the root of the inheritance tree for both run-time and user exceptions.
- `Ice.LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `Ice.UserException`
  This is the root of the inheritance tree for user exceptions.
- `Ice.TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `Ice.ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- JavaScript Mapping for Modules
- JavaScript Mapping for Built-In Types
- JavaScript Mapping for Enumerations
- JavaScript Mapping for Structures
- JavaScript Mapping for Sequences
- JavaScript Mapping for Dictionaries
- JavaScript Mapping for Constants

JavaScript Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Interfaces in JavaScript

On the client side, a Slice interface maps to a JavaScript type with methods that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Slice compiler generates code for use by the client similar to the following:

JavaScript

```
class SimplePrx extends Ice.ObjectPrx
{
    op(context){ ... }
}
```

As you can see, the compiler generates a *proxy type* `SimplePrx`. In general, the generated name is `<`*`interface-name`*`>Prx`. If an interface is nested in a module `M`, the fully-qualified name is `M.<`*`interface-name`*`>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a proxy instance. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that the prototype for `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy type defines a function with the same name. For the preceding example, we find that the operation `op` has been mapped to the function `op`. The function has an optional parameter `context` that is used by the Ice run time to store information about how to deliver a request. You normally do not need to use it. Legal values for this parameter are `null` or an instance of `Ice.Context` (which is a JavaScript Map). (We examine the `context` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

> **Asynchronous Operations**
> Unlike most other Ice language mappings, where a Slice operation generates both synchronous and asynchronous proxy methods, the JavaScript language mapping generates only an asynchronous method. Given the event-driven nature of JavaScript and its networking API, it is not feasible to provide the traditional blocking RPC model.

Client code must not create an instance of a <*interface-name*>Prx type directly. Instead, proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of null denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Interface Inheritance in JavaScript

Inheritance relationships among Slice interfaces are maintained in the generated JavaScript code. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

Given a proxy for C, a client can invoke any operation defined for interface C, as well as any operation inherited from C's base interfaces.

The Ice.ObjectPrx Interface in JavaScript

All Ice objects have Object as the ultimate ancestor type, so all proxies inherit from Ice.ObjectPrx. ObjectPrx provides a number of functions:

JavaScript

```
class ObjectPrx
{
    equals(r) { ... }
    ice_getIdentity() { ... }
    ice_isA(id, context) { ... }
    ice_ids(context) { ... }
    ice_id(context) { ... }
    ice_ping(context) { ... }
}
```

The functions behave as follows:

- **ObjectPrx.prototype.equals**
  This function compares two proxies for equality. Note that all aspects of proxies are compared by this function, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, equals returns false even though the proxies denote the same object.
- **ObjectPrx.prototype.ice_getIdentity**
  This function returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
}
```

To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

JavaScript

```
const prx1 = ...;
const prx2 = ...;
const ident1 = prx1.ice_getIdentity();
const ident2 = prx2.ice_getIdentity();

if(i1.equals(i2))
{
    // o1 and o2 denote the same object
}
else
{
    // o1 and o2 denote different objects

}
```

- **`ObjectPrx.prototype.ice_isA`**
  The `ice_isA` function determines whether the object supports a specific interface. The argument to `ice_isA` is a type ID. For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

JavaScript

```
(async function()
{
    const prx = ...;
    try
    {
        if(await prx.ice_isA("::Printer"))
        {
            // Target object supports the Printer interface!
        }
        else
        {
            // Oops, target object is a different type
        }
    }
    catch(ex)
    {
        // Handle failure
    }
}());
```

- **`ObjectPrx.prototype.ice_ids`**
  The `ice_ids` function obtains an array of strings representing all of the type IDs that the object supports.
- **`ObjectPrx.prototype.ice_id`**
  The `ice_id` function obtains the type ID of the object. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the actual type ID might be `::Base`, or it might something more derived, such as `::Derived`.
- **`ObjectPrx.prototype.ice_ping`**
  The `ice_ping` function provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExistException` or `ConnectTimeoutException`.

As remote operations, the `ice_isA`, `ice_ids`, `ice_id`, and `ice_ping` functions support an optional trailing argument representing a request context. Also note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Down-casting Proxies in JavaScript

In addition to the methods corresponding to Slice operations, the Slice-to-JavaScript compiler also generates two helper methods that support down-casting:

JavaScript

```
class SimplePrx extends Ice.ObjectPrx
{
   static checkedCast(prx, facet, context) { ... }
   static uncheckedCast(prx, facet) { ... }
}
```

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the result of the cast is a non-null reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the result of the cast is a null reference.

Note that a checked cast contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. Consequently, a checked cast is implemented as the asynchronous method `checkedCast`, which may result in a `ConnectTimeoutException` or an `ObjectNotExistException`.

Given a proxy of any type, you can use a checked cast to determine whether the corresponding object supports a given type, for example:

JavaScript

```
(async function()
{
    const prx = ...;  // Get a proxy from somewhere...
    try
    {
        const simple = await SimplePrx.checkedCast(prx);
        if(simple !== null)
        {
            // Object supports the Simple interface...
        }
        else
        {
            // Object is not of type Simple...
        }
    }
    catch(ex)
    {
        // Handle failure
    }
}());
```

In contrast, an unchecked cast does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}
interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

JavaScript

```
const obj = ...;                                  // Get proxy...
const process = ProcessPrx.uncheckedCast(obj);  // No worries...
process.launch(40, 60);                           // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

Type IDs in JavaScript

You can discover the type ID string corresponding to an interface by calling the `ice_staticId` function on the proxy type:

JavaScript

```
const id = SimplePrx.ice_staticId();
```

As an example, for an interface `Simple` in module `M`, the `ice_staticId` function returns the string `"::M::Simple"`.

Using Proxy Methods in JavaScript

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

JavaScript

```
const proxy = communicator.stringToProxy(...);
proxy = proxy.ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a checked or unchecked cast after using a factory method. The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in JavaScript

Proxies provide an `equals` function that compares proxies:

JavaScript

```
class Ice.ObjectPrx
{
    equals(other) { ... }
}
```

Note that proxy comparison with `equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `equals` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

JavaScript

```
const p1 = ...;        // Get a proxy...
const p2 = ...;        // Get another proxy...

if(!p1.equals(p2))
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function:

JavaScript

```
Ice.proxyIdentityCompare = function(lhs, rhs) { ... }
Ice.proxyIdentityAndFacetCompare = function(lhs, rhs) { ... }
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

JavaScript

```
const p1 = ...;        // Get a proxy...
const p2 = ...;        // Get another proxy...

if(Ice.proxyIdentityCompare(p1, p2) !== 0)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, -1 if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Type IDs
- JavaScript Mapping for Operations
- Request Contexts
- Operations on Object
- Proxy Methods
- Versioning

JavaScript Mapping for Operations

JavaScript's event-driven APIs makes a synchronous invocation model impractical, therefore the JavaScript language mapping provides only an asynchronous model.

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests and invocations never block. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

Basic Asynchronous API in JavaScript

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
        string getAddress(int number);
    }
}
```

Asynchronous Proxy Methods in JavaScript

`slice2js` generates the following asynchronous proxy methods:

JavaScript

```
class EmployeesPrx extends Ice.ObjectPrx
{
    getName(number, context) { ... }
    getAddress(number, context) { ... }
}
```

As you can see, a Slice operation maps to a method of the same name. (Each function accepts an optional trailing per-invocation context.)

The `getName` function, for example, sends an invocation of `getName`. Because proxy invocations do not block, the application can do other things while the operation is in progress. The application receives notification about the completion of the request by registering callbacks for success and failure.

The `getName` function, as with all functions corresponding to Slice operations, returns a value of type `Ice.AsyncResult`. This value contains the state that the Ice run time requires to keep track of the asynchronous invocation.

A proxy function has one parameter for each in-parameter of the corresponding Slice operation. For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `op` member function has the following signature:

JavaScript

```
function op(inp1, inp2, context); // Returns Ice.AsyncResult
```

The operation's return value, as well as the values of its two out-parameters, are delivered to the success callback upon completion of the request.

Passing Parameters in JavaScript

The parameter passing rules for the JavaScript mapping are very simple: parameters are passed either by value (for simple types) or by reference (for complex types). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy methods for these definitions:

JavaScript

```
class ClientToServerPrx extends Ice.ObjectPrx
{
    op1(i, f, b, s, context) { ... }
    op2(ns, ss, st, context) { ... }
    op3(proxy, context) { ... }
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

JavaScript

```
const p = ...;          // Get ClientToServerPrx proxy...

p.op1(42, 3.14, true, "Hello world!");  // Pass simple literals

const i = 42;
const f = 3.14;
const b = true;
const s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

const ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
const ss = [];
ss.push("Hello world!");
const st = new StringTable();
st.set(0, ss);
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Null Parameters in JavaScript

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Optional Parameters in JavaScript

Optional parameters use the same mapping as required parameters. The only difference is that `undefined` can be passed as the value of

an optional parameter or return value to indicate an "unset" condition. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

A client can invoke this operation as shown below:

JavaScript

```
(async function()
{
    const [retval, value] = await proxy.execute("--file log.txt");
    if(value !== undefined)
    {
        console.log("value =", value);
    }
}());
```

A well-behaved program must always compare an optional parameter to `undefined` prior to using its value.

For Slice types that support null semantics, such as proxies and objects by value, passing `null` for an optional parameter has a different meaning than passing `undefined`:

- `null` means a value was supplied for the parameter, and that value happens to be `null`
- `undefined` means no value was supplied for the parameter

`undefined` is not a legal value for required parameters.

Asynchronous Exception Semantics in JavaScript

If an invocation raises an exception, the exception is delivered to the failure callback, even if the actual error condition for the exception was encountered during the proxy function ("on the way out"). The advantage of this behavior is that all exception handling is located in the failure callback (instead of being present twice, once where the proxy function is called, and again in the failure callback).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

`AsyncResult` Type in JavaScript

The `AsyncResult` object returned by an Ice API call encapsulates the state of an asynchronous invocation. The `Ice.AsyncResult` class inherits from the standard JavaScript Promise.

> In other Ice language mappings, the `AsyncResult` type is only used for asynchronous remote invocations. In JavaScript, an `AsyncResult` object can also be returned by methods of local Ice run time objects such as `Communicator` and `ObjectAdapter` when those methods have the potential to block or internally make remote invocations.

An instance of `AsyncResult` defines the following properties:

- `communicator`
  The communicator that sent the invocation.

- `connection`
  This method returns the connection that was used for the invocation. Note that, for typical proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `connection` property only returns a non-nil value when the `AsyncResult` object is obtained by calling `flushBatchRequests` on a `Connection` object.
- `proxy`
  The proxy that was used for the invocation, or nil if the `AsyncResult` object was not obtained via a proxy invocation.
- `adapter`
  The object adapter that was used for the invocation, or nil if the `AsyncResult` object was not obtained via an object adapter invocation.
- `operation`
  The name of the operation being invoked.

`AsyncResult` objects also support the following methods:

- `AsyncResult.prototype.cancel()`
  This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. `cancel` is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning `isCompleted` returns true, and the result of the invocation is an `Ice.InvocationCanceledException`.
- `AsyncResult.prototype.isCompleted()`
  This method returns true if, at the time it is called, the result of an invocation is available. Otherwise, if the result is not yet available, the method returns false.
- `AsyncResult.prototype.isSent()`
  When you make a proxy invocation, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.
- `AsyncResult.prototype.sentSynchronously()`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).
- `AsyncResult.prototype.throwLocalException()`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.

Completion Callbacks in JavaScript

The `AsyncResult` promise returned by a proxy invocation resolves when the remote operation completes successfully or fails. The semantics of the success and failure functions depend on the proxy's invocation mode:

- Twoway proxies
  The success or failure function executes after the Ice run time in the client receives a reply from the server.
- Oneway and datagram proxies
  The success function executes after the Ice run time passes the message off to the socket. The failure function will only be called if an error occurs while Ice is attempting to marshal or send the message.
- Batch proxies
  The success function executes after the Ice run time queues the request. The failure function will only be called if an error occurs while Ice is attempting to marshal the message.

For all asynchronous Ice APIs, the failure function receives two arguments: the exception that caused the failure, and a reference to the `AsyncResult` promise for the failed invocation.

The success callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the success callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration. Ice adds a reference to the `AsyncResult` object associated with the request as the final parameter to every success callback. Recall the example we showed earlier:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The success callback for `op` will receive and array with all the parameters:

JavaScript

```
function handleSuccess(r)
{
   const [returnVal, outp1, outp2] = r;
   ...
}
```

The failure callback is invoked if the invocation fails because of an Ice run time or user exception. This callback function must have the following signature:

JavaScript

```
function handleAnException(ex)
```

For example, the code below shows how to invoke the `getName` operation:

JavaScript

```
const proxy = ...;
const empNo = ...;
proxy.getName(empNo).then(
    name => {
        console.log("Name of employee #" + empNo + " is " + name);
    }
).catch(
    ex => {
        console.log("Failed to get name of employee #" + empNo);
        console.log(ex);
    }
);
```

Let's extend the example add a call to `getAddress`:

JavaScript

```
const proxy = ...;
const empNo = ...;
proxy.getName(empNo).then(
    name => {
        console.log("Name of employee #" + empNo + " is " + name);
        return proxy.getAddress(empNo);
    }
).then(
    addr => {
        console.log("Address of employee #" + empNo + " is " + addr);
    }
).catch(
    ex => {
        console.log("failed for employee #" + empNo);
        console.log(ex);
    }
);
```

Notice here that the success callback for `getName` returns the `AsyncResult` promise for `getAddress`, which means the second success callback won't be invoked until `getAddress` completes.

Using `await` in JavaScript

While inside an asynchronous function (i.e., a function marked with the `async` keyword), you may also use the `await` keyword to give your code a control flow similar to that of a synchronous program. We can rewrite the example from the previous section as follows:

JavaScript

```
(async function()
{
    const proxy = ...;
    const empNo = ...;
    try
    {
        const name = await proxy.getName(empNo);
        console.log("Name of employee #" + empNo + " is " + name);
        const addr = await proxy.getAddress(empNo);
        console.log("Address of employee #" + empNo + " is " + addr);
    }
    catch(ex)
    {
        console.log("failed for employee #" + empNo);
        console.log(ex);
    }
}());
```

The target of the `await` keyword is a promise.

Asynchronous Oneway Invocations in JavaScript

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the proxy function on a oneway proxy for an operation that returns values or raises a user exception, the proxy function throws an instance of `TwowayOnlyException`.

The callback functions look exactly as for a twoway invocation. The failure function is only called if the invocation raised an exception during the proxy function ("on the way out"), and the success function is called as soon as the message is accepted by the local network connection. The success function takes no arguments.

Asynchronous Batch Requests in JavaScript

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the proxy function on a oneway proxy for an operation that returns values or raises a user exception, the proxy function throws an instance of `TwowayOnlyException`.

The callback functions look exactly as for a twoway invocation. The failure function is only called if the batch invocation raised an exception before being queued. The success function takes no arguments. The returned promise for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. It can also be marked completed if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The asynchronous proxy method `ice_flushBatchRequests` initiates an immediate flush of any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is accessible via the `connection` property of `AsyncResult`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

All versions of `ice_flushBatchRequests` return a promise whose success callback executes as soon as the batch request message is accepted by the local network connection. The success function takes no arguments. Ice only invokes the failure callback if an error occurs while attempting to send the message.

See Also

- Request Contexts
- Batched Invocations

JavaScript Mapping for Classes

On this page:

Basic JavaScript Mapping for Classes

A Slice class is mapped to a JavaScript class with the same name. For each Slice data member, the JavaScript instance contains a corresponding property (just as for structures and exceptions). Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

The Slice compiler generates the following code for this definition:

JavaScript

```
class TimeOfDay extends Ice.Value
{
    constructor(hour = 0, minute = 0, second = 0)
    {
        super();
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
}
```

There are a number of things to note about the generated code:

1. The generated `TimeOfDay` class inherits from `Ice.Value`. Note that `Ice.Value` is not the same as `Ice.ObjectPrx`. In other words, you cannot pass a class where a proxy is expected and vice versa.
2. The generated class provides a constructor that accepts a value for each data member.
3. The generated class defines a property for each Slice data member.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice.Value` in JavaScript

As for Slice interfaces, the generated type for a Slice class implicitly inherits from a common base type. However, the type inherits from `Ice.Value` instead of `Ice.ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`Ice.Value` defines a number of member functions:

JavaScript

```
class Ice.Value
{
    static ice_staticId() {}
    ice_id() {}
    ice_preMarshal() {}
    ice_postUnmarshal() {}
    ice_getSlicedData() {}
}
```

The member functions of `Ice.Value` behave as follows:

- `Value.ice_staticId`
  This function returns the static type ID of a class.
- `Value.prototype.ice_id`
  This function returns the actual run-time type ID for a class. If you call `ice_id` through a reference to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.
- `Value.prototype.ice_preMarshal`
  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subtype to validate its declared data members.
- `Value.prototype.ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subtype typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `Value.prototype.ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.

Class Constructors in JavaScript

The type generated for a Slice class provides a constructor that initializes each data member to a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The constructor initializes each of these data members to its declared value instead.

The constructor accepts one argument for each member of the class. This allows you to create and initialize an instance in a single statement, for example:

JavaScript

```
const tod = new TimeOfDayI(14, 45, 00); // 14:45pm
```

For derived classes, the constructor requires an argument for every member of the class, including inherited members. For example, consider the the definition from Class Inheritance once more:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}

class DateTime extends TimeOfDay
{
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
}
```

The constructors generated for these classes are similar to the following:

JavaScript

```
class TimeOfDay extends Ice.Value
{
    constructor(hour = 0, minute = 0, second = 0)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
}

class DateTime extends TimeOfDay
{
    constructor(hour, minute, second, day = 0, month = 0, year = 0)
    {
        super(hour, minute, second);
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
```

Pass `undefined` as the value of any optional data member that you wish to remain unset.

Class Data Members in JavaScript

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated type defines a corresponding property.

Optional data members use the same mapping as required data members, but an optional data member can also be set to `undefined` to indicate that the member is unset. A well-behaved program must compare an optional data member to `undefined` before using the member's value:

JavaScript

```
const v = ...
if(v.optionalMember === undefined)
{
    console.log("optionalMember is unset")
}
else
{
    console.log("optionalMember =", v.optionalMember)
}
```

Class Operations in JavaScript

> **Deprecated Feature**
> Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

With the JavaScript mapping, operations in classes are not mapped at all into the corresponding JavaScript class. The generated JavaScript class is the same whether the Slice class has operations or not.

The Slice to JavaScript compiler also generates a separate `<class-name>Disp` class, which can be used to implement an Ice object with these operations. For example:

Slice

```
class FormattedTimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;
    string format();
}
```

JavaScript

```
class FormattedTimeOfDay extends Ice.Value
{
     // ... operation format() not mapped at all here
}

// Disp class for servant implementation
class FormattedTimeOfDayDisp extends Ice.Object
{
     // ...
}
```

Value Factories in JavaScript

> While value factories are necessary when using classes with operations (a now deprecated feature), value factories may be used for any kind of class and are *not* deprecated.

Value factories allow you to create classes derived from the JavaScript class generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
interface Time
{
    TimeOfDay get();
}
```

The Ice run time will by default create and return a plain `TimeOfDay` instance.

If you wish, you can create your own custom derived class, and tell Ice to create and return these instances instead. For example:

JavaScript

```
const communicator = ...;
communicator.getValueFactoryManager().add(
    type =>
    {
        if(type === TimeOfDay.ice_staticId())
        {
            return new TimeOfDayI();
        }
        return null;
    }, TimeOfDay.ice_staticId());
```

Now, whenever the Ice run time needs to instantiate an object with the type ID `"::M::TimeOfDay"`, it calls the registered factory, which

returns a `TimeOfDayI` instance to the Ice run time.

See Also

- Classes
- Class Inheritance
- Type IDs
- Value Factories

slice2js Command-Line Options

The Slice-to-JavaScript compiler, `slice2js`, offers the following command-line options in addition to the standard options:

- `--stdout`
  Print generated code to standard output.
- `--depend-json`
  Print dependency information in JSON format to standard output by default, or to the file specified by the `--depend-file` option. No code is generated when this option is specified. The output consists of the complete list of Slice files that the input Slice files depend on through direct or indirect inclusion.

See Also

- Using the Slice Compilers

Server-Side Slice-to-JavaScript Mapping

The mapping for Slice data types to JavaScript is identical on the client side and server side. This means that everything in Client-Side Slice-to-JavaScript Mapping also applies to the server side. However, for the server side, there are a few additional things you need to know — specifically how to:

- Initialize and finalize the server-side run time
- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time

Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Although the examples we present are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any JavaScript mapping rules beyond those we described here.

> Support for server-side activities in Ice for JavaScript is currently limited to callbacks over bidirectional connections, which allows a JavaScript client to receive "push notifications" from a server. In this section, we describe the server-related activities of a JavaScript client.

Topics

Server-Side JavaScript Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing member functions in a servant class, you provide the hook that gets control flow from the Ice server-side run time into your application code.

On this page:

Skeleton Types in JavaScript

On the client side, interfaces map to proxy types. On the server side, interfaces map to *skeleton* types. A skeleton is a type that conceptually has an abstract member function for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The Slice compiler generates the following definition for this interface:

JavaScript

```
class Node extends Ice.Object
{
    constructor(){ ... }
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to JavaScript scopes with the same name, so the skeleton definitions are part of the `Filesystem` scope.
- For each Slice interface `<interface-name>`, the compiler generates a JavaScript type `<interface-name>` (`Node` in this example). This type extends `Ice.Object` and serves as the actual skeleton; it is the base type from which you derive your servant implementation.

Servant Types in JavaScript

In order to provide an implementation for an Ice object, you must create a servant type that inherits from the corresponding skeleton. For example, to create a servant for the `Node` interface, you could write:

JavaScript

```
class NodeI extends Filesystem.Node
{
    constructor(name)
    {
        this._name = name;
    }
    name(current)
    {
        return this._name;
    }
}
```

By convention, servant types have the name of their interface with an `I`-suffix, so the servant type for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant types.) Note that `NodeI` extends `Filesystem.Node`, that is, it derives from its skeleton.

Here we're using JavaScript `classes` to define our servant. But as far as Ice is concerned, the `NodeI` type must implement only a single function: the `name` method defined by the Slice interface. You can add other methods and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` method returns its value.) You can ignore the `current` parameter for now.

Normal and `idempotent` Operations in JavaScript

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

Slice

```
interface Example
{
                void   normalOp();
    idempotent void   idempotentOp();
    idempotent string readonlyOp();
}
```

The corresponding servant methods look like this:

JavaScript

```
class ExampleI extends Example
{
    normalOp(current){ ... },
    idempotentOp(current){ ... },
    readonlyOp(current){ ... }
}
```

Note that the signatures of the member functions are unaffected by the `idempotent` qualifier.

See Also

Parameter Passing in JavaScript

On this page:

Server-Side Parameters in JavaScript

For each parameter of a Slice operation, the JavaScript mapping generates a corresponding parameter for the method in the servant. In addition, every operation has an additional, trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method of the servant has a single parameter of type `Ice.Current`. We will ignore this parameter for now.

For a Slice operation that returns multiple values, the implementation returns them in an array consisting of a non-`void` return value, if any, followed by the `out` parameters in the order of declaration. An operation returning only one value simply returns the value itself.

> An operation returns multiple values when it declares multiple out parameters, or when it declares a non-`void` return type and at least one `out` parameter.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
module M
{
    interface Example
    {
        string op1(string sin);
        void op2(string sin, out string sout);
        string op3(string sin, out string sout);
    }
}
```

The servant methods would look as follows:

JavaScript

```
class ExampleI extends M.Example
{
    op1(sin, current){ ... }
    op2(sin, current){ ... }
    op3(sin, current){ ... }
}
```

The signatures of the JavaScript methods are identical because they all accept a single `in` parameter, but their implementations differ in the way they return values. For example, we could implement the operations as follows:

JavaScript

```
class ExampleI extends M.Example
{
    op1(sin, current)
    {
        console.log(sin); // In params are initialized
        return "Done";    // Return value
    }

    op2(sin, current)
    {
        console.log(sin);      // In params are initialized
        return "Hello World!"; // Out parameter
    }

    op3(sin, current)
    {
        console.log(sin); // In params are initialized
        return ["Done", "Hello World!"];
    }
}
```

Notice that `op1` and `op2` return their string values directly, whereas `op3` returns an array consisting of the return value followed by the `out` parameter.

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal JavaScript rules and do not require special-purpose API calls.

With other programming languages, you need to be mindful about potentially concurrent marshaling of objects returned by your operation implementation. There is no such concern in JavaScript since there is only one execution thread.

See Also

- Server-Side JavaScript Mapping for Interfaces
- Raising Exceptions in JavaScript
- The Current Object

Raising Exceptions in JavaScript

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

JavaScript

```
// ...
    write(text, current)
    {
        try
        {
            // Try to write file contents here...
        }
        catch(err)
        {
            const ex = new GenericError("Exception during write
operation");
            ex.ice_cause = err; // Not returned to client but may be
useful
            throw ex;
        }
    }
```

If you throw an arbitrary JavaScript error (such as an instance of `Error`), the Ice run time catches the exception and then returns an `UnknownException` to the client.

The server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw an Ice run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

See Also

- Run-Time Exceptions
- JavaScript Mapping for Exceptions
- Server-Side JavaScript Mapping for Interfaces
- Parameter Passing in JavaScript

Object Incarnation in JavaScript

Having created a servant such as the rudimentary `NodeI` type, you can instantiate the type to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a JavaScript Servant

Instantiating a servant means to allocate an instance:

JavaScript

```
const servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance.

Creating an Identity in JavaScript

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant

locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

JavaScript

```
const id = new Ice.Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

Activating a JavaScript Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

JavaScript

```
this._adapter.add(servant, id);
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object contains (among other things) the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

> Ice for JavaScript currently supports only dispatches over bidirectional connections. As a result, you never need to call `activate` on your object adapter in JavaScript; `activate` currently does nothing in JavaScript.

UUIDs as Identities in JavaScript

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. Ice for JavaScript provides a helper function that we can use to create such identities:

JavaScript

```
const uuid = Ice.generateUUID();
console.log(uuid);
```

When executed, this code prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

JavaScript

```
this._adapter.addWithUUID(new NodeI("Fred"));
```

Creating Proxies in JavaScript

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object.

At present, Ice for JavaScript only supports server-side activity over bidirectional connections. There are two ways of using bidirectional connections:

- In conjunction with a Glacier2 router, which requires very little additional code in the client aside from ensuring that the object adapter is correctly configured to use the router. This is the simplest and most common way of employing bidirectional connections.
- Manually configuring a connection for bidirectional use.

The remaining discussion in this section applies when using a Glacier2 router.

Proxies and Servant Activation in JavaScript

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

JavaScript

```
const proxy = NodePrx.uncheckedCast(this._adapter.addWithUUID(new
NodeI("Fred")));
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

Direct Proxy Creation in JavaScript

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

JavaScript

```
const id = new Ice.Identity();
id.name = Ice.generateUUID();
const o = this._adapter.createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a peer and the peer invokes an operation on the proxy, the peer will receive an `ObjectNotExistException`.

See Also

- Hello World Application
- Server-Side JavaScript Mapping for Interfaces
- Servant Locators

Asynchronous Method Dispatch (AMD) in JavaScript

JavaScript is single-threaded, therefore it is very important to consider how servant implementation decisions affect the overall application. In a JavaScript program with a graphical user interface, spending too much time in the implementation of a Slice operation can delay the processing of UI events and adversely impact the user experience.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, allows a server to receive a request but then suspend its processing. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run time.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

One use case that requires AMD is nested invocations. Since all outgoing invocations have asynchronous semantics, an operation implementation whose results depend on the outcome of a nested invocation must postpone its response until the nested invocation completes.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results and then continues to perform additional work.

On this page:

Using AMD in JavaScript

To use asynchronous dispatch, the operation implementation must return a promise object.

> The implementation can return an instance of the standard `Promise` type or any other type that provides a `then` method.

For example, suppose we define the following operation:

Slice

```
interface I
{
    int foo(short s, out long l);
}
```

The servant method for operation `foo` should have this signature:

JavaScript

```
foo(s, current) { ... }
```

It should return a promise object that, when fulfilled, returns an array containing an integer and a long.

AMD Exceptions in JavaScript

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch context (the call sequence that invokes the servant method), and the context of the promise (the call sequence that rejects the promise).

> These are not necessarily two different contexts: it is legal to return a rejected promise from the dispatch context.

It is legal for the implementation to raise an exception instead of returning a rejected promise.

AMD Example in JavaScript

To demonstrate the use of AMD in Ice, consider an operation that must make a nested invocation:

Slice

```
module Demo
{
    exception RequestFailed
    {
        string reason;
    }

    interface TaxManager
    {
        float computeTax(float amount)
            throws RequestFailed;
    }

    interface ShoppingCart
    {
        float computeTotal()
            throws RequestFailed;
    }
}
```

In this over-simplified example, a shopping cart object must query a tax manager object in order to compute the total amount owed by the customer.

Our servant class derives from `Demo.ShoppingCart` and supplies a definition for the `computeTotal` method:

JavaScript

```
class ShoppingCartI extends Demo.ShoppingCart
{
    constructor()
    {
        this._subtotal = 0.0;
    }

    computeTotal(current)
    {
        const taxManager = ... // get TaxManager proxy
        const subtotal = this._subtotal;
        return taxManager.computeTax(subtotal).then(
            tax => subtotal + tax;
            ex => {
                if(ex instanceof Demo.RequestFailed)
                {
                    throw ex; // Relay RequestFailed exception from
computeTax
                }
                else
                {
                    throw new Demo.RequestFailed("failed: " +
ex.toString());
                }
            });
    }
}
```

The implementation of `computeTotal` makes a nested invocation of `computeTax and returns a promise whose` resolve and reject functions return the result of the `computeTotal` dispatch.

See Also

- User Exceptions
- JavaScript Mapping for Operations

Slice-to-JavaScript Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to JavaScript is identical to the mapping for these constructs without the `local` qualifier. The generated JavaScript code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

Data members on local Slice types (classes, exceptions and structs) are mapped to JavaScript just like the data members of the corresponding non-local Slice construct.

A `local interface` has no mapping in JavaScript since the language is loosely typed and has no equivalent for an interface.

An operation defined in a `local class` or `local interface` generates no code in JavaScript. A method implementation has the same signature as the client-side mapping without the trailing `Context` parameter.

The rest of this section describes the mapping of the remaining local types to JavaScript:

- JavaScript Mapping for Local Classes
- JavaScript Mapping for Local Exceptions

JavaScript Mapping for Local Classes

On this page:

Mapped JavaScript Class

A local Slice class is mapped to a JavaScript class with the same name. For example:

Slice

```
module M
{
    local class Example
    {
        ...
    }
}
```

is mapped to the JavaScript class `Example`:

JavaScript

```
class Example
{
    ...
};
```

LocalObject in JavaScript

Local Slice classes implicitly derive from `LocalObject`, which is mapped to the native `Object` class in JavaScript.

Mapping for Local Interface Inheritance in JavaScript

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` is mapped to class inheritance in JavaScript, but `implements` is not mapped. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

JavaScript

```
// No mapping for A or B
class C
{
    ...
};
class D extends C
{
    ...
};
```

JavaScript Mapping for Local Exceptions

On this page:

Mapped JavaScript Class

A local Slice exception is mapped to a JavaScript class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the JavaScript class `InitializationException`:

JavaScript

```
class InitializationException extends Ice.LocalException
{
    ...
};
```

Base Class for Local Exceptions in JavaScript

All mapped JavaScript classes for local exceptions extend the class `Ice.LocalException`:

JavaScript

```
class LocalException extends Exception
{
    ...
};
```

`LocalException` derives from `Ice.Exception`, which derives from JavaScript's native `Error` class.

Mapping for Local Exception Inheritance in JavaScript

A local Slice exception can extend another Slice exception, which is mapped to class inheritance in JavaScript. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

JavaScript

```
class ErrorBase extends Ice.LocalException
{
    ...
};
class ResourceError extends ErrorBase
{
    ...
};
```

MATLAB Mapping

Ice currently provides a client-side mapping for MATLAB, but not a server-side mapping.

Topics

Initialization in MATLAB

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `Ice::Communicator` object.

A `Communicator` is a local MATLAB object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application.

You initialize the Ice run time by calling `Ice.initialize`, for example:

MATLAB

```
[communicator, remainingArgs] = Ice.initialize(args);
```

`Ice.initialize` accepts an optional argument list. The function scans the argument list for any command-line options that are relevant to the Ice run time; any such options are removed from the argument list so, when `Ice.initialize` returns, the only options and arguments remaining in `remainingArgs` are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your program, you must call `Communicator.destroy`. The `destroy` method is responsible for finalizing the Ice run time. In particular, `destroy` ensures that any outstanding threads started by the underlying Ice C++ run-time are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your program to terminate without calling `destroy` first.

The general shape of our Ice MATLAB application is therefore:

MATLAB

```
communicator = Ice.initialize(args); % here we ignore the remaining args
cleanup = onCleanup(@() communicator.destroy());
```

> You can safely call `destroy` multiple times on a communicator. `destroy` does not throw any exception.

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-MATLAB Mapping

The client-side Slice-to-MATLAB mapping defines how Slice data types are translated to MATLAB types, and how clients invoke operations, pass parameters, and handle errors. Much of the MATLAB mapping is intuitive. For example, Slice sequences map to MATLAB arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in MATLAB.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the MATLAB mapping, you should need no more than the Slice definition of your application and knowledge of the MATLAB mapping rules. In particular, looking through the generated code in order to discern how to use the MATLAB mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

The `Ice` Module

All of the APIs for the Ice run time are nested in the `Ice` package, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` package are generated from Slice definitions; other parts of the `Ice` package provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` package throughout the remainder of the chapter.

A MATLAB application can load the Ice run time using the `loadlibrary` function:

```
loadlibrary('ice', @iceproto);
```

or

```
loadlibrary('ice'); % requires a C/C++ compiler to build the
generated "thunk" library
```

If the function executes without error, the Ice run time is loaded and available for use. You can determine the version of the Ice run time you have just loaded by calling the `stringVersion` function:

```
icever = Ice.stringVersion();
```

Topics

MATLAB Mapping for Identifiers

A Slice identifier maps to an identical MATLAB identifier. For example, the Slice identifier `Clock` becomes the MATLAB identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a MATLAB keyword, the corresponding MATLAB identifier is suffixed with an underscore. For example, the Slice identifier `while` is mapped as `while_`.

> You should try to avoid such identifiers as much as possible.

See Also

- Lexical Rules

MATLAB Mapping for Modules

A Slice module maps to a MATLAB package with the same name. The mapping preserves the nesting of the Slice definitions.

See Also

- Modules

MATLAB Mapping for Basic Types

On this page:

Mapping of Slice Built-In Types to MATLAB Types

The Slice built-in types are mapped to MATLAB types as shown in this table:

Slice	MATLAB
`bool`	`logical`
`byte`	`uint8`
`short`	`int16`
`int`	`int32`
`long`	`int64`
`float`	`single`
`double`	`double`
`string`	character vector (1-by-n array of `char`), empty 0-by-0 character array for empty strings

String Mapping in MATLAB

String values returned as the result of a Slice operation, including return values, out parameters, and data members, are converted from UTF-8 (Ice's on-the-wire encoding) to MATLAB's native UTF-16 encoding.

The MATLAB mapping accepts empty arrays in addition to character vectors as `string` input values for remote Slice operations; each occurrence of an empty array is marshaled as an empty string. Ice assumes that all character vectors contain valid UTF-16 encoded strings and converts them to UTF-8 prior to marshaling.

> The MATLAB mapping currently does not allow you to map Slice `strings` to the MATLAB `string` type added in MATLAB R2016b.

See Also

- Basic Types

MATLAB Mapping for Enumerations

A Slice enumeration maps to the corresponding enumeration in MATLAB. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The MATLAB mapping for `Fruit` is shown below:

MATLAB

```
classdef Fruit < int32
    enumeration
        Apple (0)
        Pear (1)
        Orange (2)
    end
    methods(Static)
        ...
        function r = ice_getValue(v)
            switch v
                case 0
                    r = Fruit.Apple;
                case 1
                    r = Fruit.Pear;
                case 2
                    r = Fruit.Orange;
                otherwise
                    throw(Ice.MarshalException(...));
            end
        end
    end
end
```

Given the above definitions, we can use enumerated values as follows:

MATLAB

```
f1 = Fruit.Apple;
f2 = Fruit.Orange;

if f1 == Fruit.Apple % Compare with constant
    % ...
end

if f1 == f2          % Compare two enums
    % ...
end

switch f2            % Switch on enum
    case Fruit.Apple
        % ...
    case Fruit.Pear
        % ...
    case Fruit.Orange
        % ...
end
```

You can obtain the ordinal value of an enumerator using the `int32` function:

MATLAB

```
val = int32(Fruit.Pear);
assert(val == 1);
```

To convert an integer into its equivalent enumerator, call the `ice_getValue` function:

MATLAB

```
f = Fruit.ice_getValue(2);
assert(f == Fruit.Orange);
```

This function raises an exception if the given integer does not match any of the enumerators.

Note that the generated class contains a number of other members, which we have not shown. These members are internal to the Ice run time and you must not use them in your application code (because they may change from release to release).

The `Fruit` definition above shows the ordinal values assigned by default to the enumerators. Suppose we modify the definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The generated code changes accordingly:

MATLAB

```
classdef Fruit < int32
    enumeration
        Apple (0)
        Pear (3)
        Orange (4)
    end
    ...
end
```

See Also

- Enumerations

MATLAB Mapping for Structures

On this page:

Basic MATLAB Mapping for Structures

A Slice structure maps to a MATLAB value class containing a public property for each member of the structure. For example, here is our `Employee` structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The MATLAB mapping generates the following definition for this structure:

MATLAB

```
classdef Employee
    properties
        number int64
        firstName char
        lastName char
    end
    methods
        function obj = Employee(number, firstName, lastName)
            ...
        end
    end
    ...
end
```

The class provides a constructor whose optional arguments correspond to the data members. This allows you to instantiate and initialize the class in a single statement (instead of having to first instantiate the class and then assign to its members). You must either call the constructor with no arguments or with arguments for all of the data members.

Default Property Values

Calling the generated constructor with no arguments assigns a default value appropriate for each member's type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration

`struct`	Default-constructed value
Numeric	Zero
`bool`	`false`
`sequence`	Empty array
`dictionary`	Instance of the mapped type
`class`	Empty array

You can also declare different default values for members of primitive and enumerated types.

See Also

- Structures

MATLAB Mapping for Sequences

The MATLAB mapping for a Slice sequence depends on the element type of the sequence:

Element Type	Mapped Sequence Type
`enum`, `struct`, basic type other than `string`	vector[1] of the mapped element type, empty 0-by-0 array of the mapped element type for empty sequence
all other types: `class`, `dictionary`, `proxy`, `sequence`, `string`	1-by-n cell array of the mapped element type, empty 0-by-0 cell array for empty sequence

> For sequences of user-defined types, the mapping generates a class with the same name as the sequence. This class contains only marshaling code and therefore we do not describe it here.

See Also

- Sequences

[1] These vectors are 1-by-n arrays, where n is the size of the sequence.

MATLAB Mapping for Dictionaries

There are two mappings for a Slice dictionary, depending on its key type:

Key Type	Value Type	Mapped Type	Comments
Integral type (`short`, `int`, `long`), `bool`, `byte`, `enum`, `string`	Any	`containers.Map`	Enumeration keys must be converted to integers.
`struct`	Any	structure array	`containers.Map` does not support user-defined key types.

The mapping generates a class with the same name as the dictionary. This class provides one useful function for applications: a `new` function that returns an empty instance of the mapped type.

On this page:

Mapping to `containers.Map`

Consider the definition of our EmployeeMap once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

Since the key is a primitive type, `EmployeeMap` maps to `containers.Map`. We can use it as shown below:

MATLAB

```
em = containers.Map('KeyType', 'int64', 'ValueType', 'any');

e = Employee();
e.number = 31;
e.firstName = 'James';
e.lastName = 'Gosling';

em(e.number) = e;
```

The value for `KeyType` and `ValueType` depends on the corresponding Slice type as shown in the following table:

Slice Type	Corresponding KeyType	Corresponding ValueType
`bool`	`int32`	`logical`
`byte`	`int32`	`uint8`
`short`	`int32`	`int16`
`int`	`int32`	`int32`
`long`	`int64`	`int64`
`enum`	`int32`	`any`
`string`	`char`	`char`

`float`		`single`
`double`		`double`
any other type		`any`

Mapping to Structure Array

A dictionary with a structure key type maps to a MATLAB structure array. This mapping sacrifices some functionality, most notably the efficient indexing provided by `containers.Map`, but it's a straightforward mapping that utilizes the native MATLAB structure type.

Consider this example:

Slice

```
struct ID
{
    string symbol;
    string exchange;
}

dictionary<ID, double> Quotes;
```

We can construct this dictionary in MATLAB as follows:

MATLAB

```
quotes = [];
quotes(1).key = ID('CVX', 'NYSE');
quotes(1).value = 100.00;
```

Each element of the structure array must have `key` and `value` fields whose types correspond to the mapped types for the Slice dictionary's key and value.

An empty dictionary with a `struct` key type is mapped to an empty structure array with fields named `key` and `value`.

See Also

- Dictionaries

MATLAB Mapping for Constants

Here are the sample constant definitions once more:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

MATLAB

```
classdef AppendByDefault
    properties(Constant)
        value logical = true
    end
end

classdef LowerNibble
    properties(Constant)
        value uint8 = 15
    end
end

classdef Advice
    properties(Constant)
        value char = 'Don''t Panic!'
    end
end

classdef TheAnswer
    properties(Constant)
        value int16 = 42
    end
end

classdef PI
    properties(Constant)
        value double = 3.1416
    end
end

classdef FavoriteFruit
    properties(Constant)
        value = Fruit.Pear
    end
end
```

As you can see, each Slice constant is mapped to a MATLAB class with the same name as the constant. The class contains a constant property named `value` that holds the value of the constant.

Slice string literals that contain non-ASCII characters or universal character names are mapped to MATLAB string literals with UTF-16 character codes. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

MATLAB

```
classdef Egg
    properties(Constant)
        value char = sprintf('\x0153uf')
    end
end

classdef Heart
    properties(Constant)
        value char = sprintf('c\x0153ur')
    end
end

classdef Banana
    properties(Constant)
        value char = sprintf('\xd83c\xdf4c')
    end
end
```

The mapping uses the `sprintf` function to convert encoded strings into native MATLAB character arrays.

See Also

- Constants and Literals

MATLAB Mapping for Exceptions

On this page:

Base Exception Classes

The class `Ice.Exception` is the root of the derivation tree for Ice exceptions:

MATLAB

```
classdef (Abstract) Exception < MException
    methods(Abstract)
        ice_id(obj)
    end
    methods
        function obj = Exception(id, msg)
            obj = obj@MException(id, msg)
        end
    end
end
```

The `ice_id` function returns the type ID of the exception. For example, if you call the `ice_id` member function of a `BadZoneName` exception defined in module `M`, it returns the string `'::M::BadZoneName'`. The constructor accepts the standard arguments for MATLAB's base exception class `MException` and passes those arguments along to the base class constructor.

The class `Ice.UserException` derives from `Ice.Exception` and serves as the base type for application-defined Slice exceptions:

MATLAB

```
classdef (Abstract) UserException < Ice.Exception
    methods
        function obj = UserException(id, msg)
            obj = obj@Ice.Exception(id, msg)
        end
    end
end
```

Finally, the class `Ice.LocalException` also derives from `Ice.Exception`. All of the Ice run time's local exceptions derive from `Ice.LocalException`:

MATLAB

```
classdef (Abstract) LocalException < Ice.Exception
    methods
        function obj = LocalException(id, msg)
            obj = obj@Ice.Exception(id, msg)
        end
    end
end
```

MATLAB Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

MATLAB

```
classdef GenericError < Ice.UserException
    properties
        reason char
    end
    methods
        function obj = GenericError(ice_exid, ice_exmsg, reason)
            ...
        end
        function id = ice_id(obj)
            ...
        end
    end
    ...
end

classdef BadTimeVal < GenericError
    methods
        function obj = BadTimeVal(ice_exid, ice_exmsg, reason)
            ...
        end
        function id = ice_id(obj)
            ...
        end
    end
    ...
end

classdef BadZoneName < GenericError
    methods
        function obj = BadZoneName(ice_exid, ice_exmsg, reason)
            ...
        end
        function id = ice_id(obj)
            ...
        end
    end
    ...
end
```

Each Slice exception is mapped to a MATLAB class with the same name. For each data member, the corresponding class contains a public property. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have properties.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception also defines an `ice_id` method, which returns the Slice type ID of the exception.

All user exceptions are derived from the base class `UserException`. This allows you to handle all user exceptions generically by testing

whether an instance is-a `UserException`. `UserException`, in turn, derives from `Exception`, which derives from MATLAB's native `MException` class.

Note that the generated exception classes contain other methods that are not shown. However, those methods are internal to the MATLAB mapping and are not meant to be called by application code.

Here's an example that shows how we could handle these exceptions:

MATLAB

```
try
    % ...
catch ex
    if isa(ex, 'BadZoneName')
        % handle BadZoneName
    elseif isa(ex, 'BadTimeVal')
        % handle BadTimeVal
    elseif isa(ex, 'GenericError')
        % handle GenericError
    else
        % Allow any other exception to propagate
        rethrow(ex);
    end
end
```

Constructing a User Exception

The first two arguments for every exception constructor are an identifier and a message; these arguments are passed up the inheritance hierarchy to the `MException` class. You can pass empty strings for these arguments and the constructor will supply default values.

If an exception declares or inherits any data members, the constructor accepts one additional parameter for each data member so that you can construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members). For a derived exception, the constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

You must either call the constructor with no arguments or with arguments for all of the parameters.

Calling the constructor with no arguments assigns a default value appropriate for each member's type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	`false`
`sequence`	Empty array
`dictionary`	Instance of the mapped type
`class`	Empty array

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

Optional Data Members

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice.Unset` to indicate that the member is unset. A well-behaved program must test an optional data member before using its value:

MATLAB

```
try
    ...
catch ex
    if ex.optionalMember ~= Ice.Unset
        fprintf('optionalMember = %s\n', ex.optionalMember);
    else
        fprintf('optionalMember is unset\n');
    end
end
```

The `Ice.Unset` marker value has different semantics than an empty array. Since an empty array is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to an empty array is considered to be set. If you need to distinguish between an unset value and a value set to an empty array, you can do so as follows:

MATLAB

```
try
    ...
catch ex
    if ex.optionalMember == Ice.Unset
        fprintf('optionalMember is unset\n');
    elseif isempty(ex.optionalMember)
        fprintf('optionalMember is empty\n');
    else
        fprintf('optionalMember = %s\n', ex.optionalMember);
    end
end
```

MATLAB Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `LocalException` (which, in turn, derives indirectly from `MException`).

Recall the inheritance diagram for user and run-time exceptions. By testing exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `UserException`
  This is the root of the inheritance tree for user exceptions.
- `TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `MException`.

You will probably have little need to test run-time exceptions for their most-derived type and instead test them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to handle explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions

MATLAB Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Interfaces in MATLAB

A Slice interface maps to a MATLAB class with methods that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Slice compiler generates the following definition for use by the client:

MATLAB

```
classdef SimplePrx < Ice.ObjectPrx
    methods
        function op(obj, varargin)
            ...
        end
    end
    methods(Static)
        function id = ice_staticId()
            ...
        end
        function r = checkedCast(proxy, varargin)
            ...
        end
        function r = uncheckedCast(proxy, varargin)
            ...
        end
    end
    ...
end
```

As you can see, the compiler generates a *proxy class* `SimplePrx`. In general, the generated name is `<`*`interface-name`*`>Prx`. If an interface is nested in a module `M`, the generated class is part of package `M`, so the fully-qualified name is `M.<`*`interface-name`*`>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a proxy instance. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. For the preceding example, we find that the operation `op` has been mapped to the method `op`. Notice that the `op` method accepts a variable argument list. This allows you to specify an optional request context of type `containers.Map`. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it.

Proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

An empty array denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Interface Inheritance in MATLAB

Inheritance relationships among Slice interfaces are maintained in the generated MATLAB classes. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

The generated code for `CPrx` reflects the inheritance hierarchy:

MATLAB

```
classdef CPrx < APrx & BPrx
    ...
end
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

The `ObjectPrx` Interface in MATLAB

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `ObjectPrx`. `ObjectPrx` provides a number of methods:

MATLAB

```
classdef ObjectPrx < handle
    methods
        function r = eq(obj, other)
        function r = ice_getIdentity(obj)
        function r = ice_isA(obj, id, varargin)
        function r = ice_ids(obj, varargin)
        function r = ice_id(obj, varargin)
        function ice_ping(obj, varargin)
        % ...
    end
end
```

The methods behave as follows:

- **`eq`**
  This method allows two proxies to be compared for equality using the == operator. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

  ```
  module Ice
  {
      struct Identity
      {
          string name;
          string category;
      }
  }
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

MATLAB

```
proxy1 = ...;
proxy2 = ...;
id1 = proxy1.ice_getIdentity();
id2 = proxy2.ice_getIdentity();

if isequal(id1, id2)
    % proxy1 and proxy2 denote the same object
else
    % proxy1 and proxy2 denote different objects
end
```

- **ice_isA**
  The ice_isA method determines whether the object denoted by the proxy supports a specific interface. The argument to ice_isA is a type ID. For example, to see whether a proxy of type ObjectPrx denotes a Printer object, we can write:

  MATLAB

  ```
  proxy = ...;
  if ~isempty(proxy) && proxy.ice_isA("::Printer")
      % o denotes a Printer object
  else
      % o denotes some other type of object
  end
  ```

  Note that we are testing whether the proxy is null before attempting to invoke the ice_isA method. This avoids getting an error if the proxy is an empty array.

- **ice_ids**
  The ice_ids method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- **ice_id**
  The ice_id method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type BasePrx, with a static type ID of ::Base, the return value of ice_id might be ::Base, or it might something more derived, such as ::Derived.

- **ice_ping**
  The ice_ping method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

The ice_isA, ice_ids, ice_id, and ice_ping methods are remote operations and therefore support an optional request context. Also note that there are other methods in ObjectPrx, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Proxy Helper Methods in MATLAB

As we saw earlier, the Slice-to-MATLAB compiler generates static helper methods that support down-casting and type discovery:

MATLAB

```
classdef SimplePrx < Ice.ObjectPrx
    ...
    methods(Static)
        function id = ice_staticId()
            ...
        end
        function r = checkedCast(proxy, varargin)
            ...
        end
        function r = uncheckedCast(proxy, varargin)
            ...
        end
    end
    ...
end
```

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a non-null reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns an empty array. You can optionally supply a facet and a request context, in that order; if you supply a request context, you must also supply a facet.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

MATLAB

```
proxy = ...;        % Get a proxy from somewhere...

simple = SimplePrx.checkedCast(proxy);
if ~isempty(simple)
    % Object supports the Simple interface...
else
    % Object is not of type Simple...
end
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

MATLAB

```
proxy = ...;                                % Get proxy...
process = ProcessPrx.uncheckedCast(proxy);  % No worries...
process.launch(40, 60);                     % Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

You may optionally supply a facet argument to `uncheckedCast`.

Another method generated for every interface is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Using Proxy Methods in MATLAB

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

MATLAB

```
proxy = communicator.stringToProxy(...);
proxy = proxy.ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a `checkedCast` or `uncheckedCast` after using a factory method. Furthermore, the mapping generates type-specific factory methods so that no casts are necessary:

MATLAB

```
base = communicator.stringToProxy(...);
hello = HelloPrx.checkedCast(base);
hello = hello.ice_invocationTimeout(10000); % No cast is necessary
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type using `checkedCast` or `uncheckedCast`.

Object Identity and Proxy Comparison in MATLAB

Proxies can be compared for equality using the == operator. Note that proxy comparison with == uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with == tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

MATLAB

```
proxy1 = ...;        % Get a proxy...
proxy2 = ...;        % Get another proxy...

if proxy1 == proxy2
    % proxy1 and proxy2 denote the same object         % Correct
else
    % proxy1 and proxy2 denote different objects       % WRONG!
end
```

Even though `proxy1` and `proxy2` differ, they may still denote the same Ice object. This can happen because, for example, both `proxy1` and `proxy2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` package:

MATLAB

```
function r = proxyIdentityCompare(lhs, rhs)
function r = proxyIdentityAndFacetCompare(lhs, rhs)
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

MATLAB

```
proxy1 = ...;        % Get a proxy...
proxy2 = ...;        % Get another proxy...

if Ice.proxyIdentityCompare(p1, p2) == 0
    % proxy1 and proxy2 denote the same object         % Correct
else
    % proxy1 and proxy2 denote different objects       % Correct
end
```

The function returns 0 if the identities are equal, `-1` if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Type IDs
- Request Contexts
- Operations on Object
- Proxy Methods
- Versioning

MATLAB Mapping for Operations

On this page:

Basic MATLAB Mapping for Operations

As we saw in the mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

MATLAB

```
node = ...;             % Initialize proxy
name = node.name();     % Get name via RPC
```

Normal and `idempotent` Operations in MATLAB

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent  string op2();
}
```

The proxy interface for this is:

MATLAB

```
classdef ExamplePrx < Ice.ObjectPrx
    function r = op1(obj, varargin)
    function r = op2(obj, varargin)
end
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

Passing Parameters in MATLAB

In Parameters in MATLAB

The parameter passing rules for the MATLAB mapping are very simple: parameters are passed either by value (for value types) or by reference (for handle types). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy for these definitions:

MATLAB

```
classdef ClientToServerPrx < Ice.ObjectPrx
    methods
        function op1(obj, i, f, b, s, varargin)
            ...
        end
        function op2(obj, ns, ss, st, varargin)
            ...
        end
        function op3(obj, proxy, varargin)
            ...
        end
    end
end
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

MATLAB

```
p = ...;                                % Get ClientToServerPrx proxy...

p.op1(42, 3.14f, true, 'Hello world!'); % Pass simple literals

i = 42;
f = 3.14;
b = true;
s = 'Hello world!';
p.op1(i, f, b, s);                      % Pass simple variables

ns = NumberAndString();
ns.x = 42;
ns.str = 'The Answer';
ss = { 'Hello world!' };
st = StringTable.new();
st(0) = ns;
p.op2(ns, ss, st);                      % Pass complex variables

p.op3(p);                               % Pass proxy
```

Out Parameters in MATLAB

The MATLAB mapping uses the conventional language mechanism for returning one or more result values. Here are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Slice compiler generates the following code for these definitions:

MATLAB

```
classdef ServerToClientPrx < Ice.ObjectPrx
    methods
        function [i, f, b, s] = op1(obj, varargin)
            ...
        end
        function [ns, ss, st] = op2(obj, varargin)
            ...
        end
        function proxy = op3(obj, varargin)
            ...
        end
    end
end
```

Null Parameters in MATLAB

Slice classes and proxies naturally have "empty" or "not there" semantics, but other types such as sequences, dictionaries, and strings do not have the concept of a null value. To make life with these types easier, whenever you pass an empty array (`[]`) as a parameter of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending it. Note that using null parameters in this way does *not* create null semantics for Slice

sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as an empty array or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Optional Parameters in MATLAB

Optional parameters use the same mapping as required parameters. The only difference is that `Ice.Unset` can be passed as the value of an optional parameter or return value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

A client can invoke this operation as shown below:

MATLAB

```
[i, v] = proxy.execute('--file log.txt');
[i, v] = proxy.execute(Ice.Unset);

if v ~= Ice.Unset
    fprintf('value = %f\n', v); % v is set to a value
end
```

A well-behaved program must always test an optional parameter prior to using its value. Keep in mind that the `Ice.Unset` marker value has different semantics than an empty array. Since an empty array is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional parameter is set. An optional parameter set to an empty array is considered to be set. If you need to distinguish between an unset parameter and a parameter set to an empty array, you can do so as follows:

MATLAB

```
if optionalParam == Ice.Unset
    fprintf('optionalParam is unset\n');
elseif isempty(optionalParam)
    fprintf('optionalParam is empty\n');
else
    fprintf('optionalParam = %s\n', optionalParam);
end
```

Exception Handling in MATLAB

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as MATLAB exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

MATLAB

```
child = ...;   % Get child proxy...

try
    child.askToCleanUp();
catch ex
    if isa(ex, 'Tantrum')
        fprintf('The child says: %s\n', ex.reason);
    else
        rethrow(ex);
    end
end
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

MATLAB

```
function client(args)
    try
        child = ...;   % Get child proxy...
        try
            child.askToCleanUp();
            child.praise();             % Give positive feedback...
        catch ex
            if isa(ex, 'Tantrum')
                fprintf('The child says: %s\n', ex.reason);
                child.scold();          % Recover from error...
            else
                rethrow(ex);
            end
        end
    catch ex
        if isa(ex, 'Ice.UserException')
            % handle uncaught user exception
        elseif isa(ex, 'Ice.LocalException')
            % handle uncaught local exception
        else
            % other exception type
            rethrow(ex);
        end
    end
end
```

See Also

- Operations

MATLAB Mapping for Classes

On this page:

Basic MATLAB Mapping for Classes

A Slice class is mapped to a MATLAB class with the same name. The generated class contains a public property for each Slice data member (just as for structures and exceptions). Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

The Slice compiler generates the following code for this definition:

MATLAB

```
classdef TimeOfDay < Ice.Value
    properties
        hour int16
        minute int16
        second int16
        tz char
    end
    methods
        function obj = TimeOfDay(hour, minute, second, tz)
            ...
        end
        function id = ice_id(obj)
            ...
        end
        ...
    end
    methods(Static)
        function id = ice_staticId()
            ...
        end
    end
end
```

There are a several things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice.Value`. This means that all classes implicitly inherit from `Value`, which is the ultimate ancestor of all classes.
2. The generated class contains a public property for each Slice data member.
3. The generated class has a constructor that optionally takes one argument for each data member.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice::Value` in MATLAB

Classes implicitly inherit from a common base class, `Value`, which is mapped to `Ice.Value`. `Value` is a very simple base class with just a few methods:

MATLAB

```
classdef (Abstract) Value < matlab.mixin.Copyable
    methods
        function ice_preMarshal(obj)
        end
        function ice_postUnmarshal(obj)
        end
        function r = ice_getSlicedData(obj)
            ...
        end
        ...
    end
    methods(Abstract)
        id = ice_id(obj)
    end
    methods(Static)
        function id = ice_staticId()
            id = '::Ice::Object'
        end
    end
end
```

`Value` derives from `matlab.mixin.Copyable`, which means subclasses are handle types and instances can be copied using the `copy` function.

The `Value` methods behave as follows:

- `ice_preMarshal`
  The Ice run time invokes this method prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this method after unmarshaling an object's state. A subclass typically overrides this method when it needs to perform additional initialization using the values of its declared data members.
- `ice_id`
  This method returns the actual run-time type ID for a class instance. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during unmarshaling or an empty array otherwise.

Class Data Members in MATLAB

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public property.

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice.Unset` to indicate that the member is unset. A well-behaved program must test an optional data member before using its value:

MATLAB

```
obj = ...;
if obj.optionalMember ~= Ice.Unset
    fprintf('optionalMember = %s\n', ex.optionalMember);
else
    fprintf('optionalMember is unset\n');
end
```

The `Ice.Unset` marker value has different semantics than an empty array. Since an empty array is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to an empty array is considered to be set. If you need to distinguish between an unset value and a value set to an empty array, you can do so as follows:

MATLAB

```
obj = ...;
if obj.optionalMember == Ice.Unset
    fprintf('optionalMember is unset\n');
elseif isempty(obj.optionalMember)
    fprintf('optionalMember is empty\n');
else
    fprintf('optionalMember = %s\n', ex.optionalMember);
end
```

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the property with protected visibility. As a result, the property can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    ["protected"] string tz;    // e.g. GMT, PST, EDT...
}
```

The Slice compiler produces the following generated code for this definition:

MATLAB

```
classdef TimeOfDay < Ice.Value
    properties(Access=protected)
        hour int16
        minute int16
        second int16
        tz char
    end
    ...
end
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;          // e.g. GMT, PST, EDT...
}
```

Value Factories in MATLAB

Value factories allow you to create classes derived from the MATLAB classes generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
interface Time
{
    TimeOfDay get();
}
```

The default behavior of the Ice run time will create and return an instance of the generated `TimeOfDay` class.

If you wish, you can create your own custom derived class, and tell Ice to create and return these instances instead. For example:

MATLAB

```
classdef CustomTimeOfDay < TimeOfDay
    methods
        function format(obj)
            % prints formatted data members
        end
    end
end
```

You then create and register a value factory for your custom class with your Ice communicator:

MATLAB

```
function v = factory(type)
    assert(strcmp(type, TimeOfDay.ice_staticId()));
    v = CustomTimeOfDay();
end

communicator = ...;
communicator.getValueFactoryManager().add(@factory,
TimeOfDay.ice_staticId());
```

Class Constructors in MATLAB

If a class declares or inherits any data members, the generated constructor accepts one parameter for each data member so that you can construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members). For a derived class, the constructor accepts one argument for each base class member, plus one argument for each derived class member, in base-to-derived order.

You must either call the constructor with no arguments or with arguments for all of the parameters.

Calling the constructor with no arguments assigns a default value appropriate for each member's type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	`false`
`sequence`	Empty array
`dictionary`	Instance of the mapped type
`class`	Empty array

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

See Also

Asynchronous Method Invocation (AMI) in MATLAB

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the application. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

Asynchronous API in MATLAB

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in MATLAB

In addition to the synchronous proxy method, `slice2matlab` generates the following asynchronous proxy method:

MATLAB

```
classdef EmployeesPrx < Ice.ObjectPrx
    methods
        function result = getName(obj, number, varargin)  % Synchronous
method
            ...
        end
        function future = getNameAsync(obj, number, varargin) %
Asynchronous method
            ...
        end
    end
    ...
end
```

As you can see, the `getName` Slice operation generates a `getNameAsync` method that optionally accepts a per-invocation context.

The `getNameAsync` method sends (or queues) an invocation of `getName`. This method does not block the application. It returns an instance of `Ice.Future` that you can use in a number of ways, including blocking to obtain the result, querying its state, and canceling the invocation.

Here's an example that calls `getNameAsync`:

MATLAB

```
e = ...; % Get EmployeesPrx proxy
future = e.getNameAsync(99);

% Continue to do other things here...

name = f.fetchOutputs();
```

Because `getNameAsync` does not block, the application can do other things while the operation is in progress.

An asynchronous proxy method uses the same parameter mapping as for synchronous operations; the only difference is that the result (if any) is obtained from the future. For example, consider the following operation:

Slice

```
interface Example
{
    double op(int inp1, string inp2, out bool outp1, out long outp2);
}
```

The generated code looks like this:

MATLAB

```
classdef ExamplePrx < Ice.ObjectPrx
    methods
        function future = opAsync(obj, inp1, inp2, varargin)
            ...
        end
        ...
    end
    ...
end
```

Now let's call `fetchOutputs` to demonstrate how to retrieve the results when the invocation completes:

MATLAB

```
e = ...; % Get EmployeesPrx proxy
future = e.opAsync(5, 'demo');
...
[r, outp1, outp2] = future.fetchOutputs();
```

Asynchronous Exception Semantics in MATLAB

If an invocation raises an exception, the exception will be thrown when the application calls `fetchOutputs` on the future. The exception is provided by the future, even if the actual error condition for the exception was encountered during the call to the `opAsync` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that handles the future (instead of being present twice, once where the `opAsync` method is called, and again where the future is handled).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

`Future` Class in MATLAB

The `Future` object that is returned by asynchronous proxy methods has an API that resembles MATLAB's `parallel.Future` class:

MATLAB

```
classdef Future < ...
    methods
        function ok = wait(obj, state, timeout)
        function varargout = fetchOutputs(obj)
        function cancel(obj)
    end
    properties(SetAccess=private) % Read only properties
        ID
        NumOutputArguments
        Operation
        Read
        State
    end
end
```

The members have the following semantics:

- `wait()`
  This method blocks until the invocation completes and returns `true` if it completed successfully or `false` if it failed.
- `wait(state)`
  This method blocks until the desired state is reached (see the description of the `State` property below). For example, calling `future.wait('finished')` is equivalent to calling `future.wait()`. The method returns `true` if the desired state was reached and no exception has occurred, or `false` otherwise.
- `wait(state, timeout)`

This method blocks for a maximum of `timeout` seconds until the desired state is reached, where `timeout` is a double value. The method returns `true` if the desired state was reached and no exception has occurred, or `false` otherwise.

- `fetchOutputs()`
  This method blocks until the invocation completes. If it completed successfully, `fetchOutputs` returns the results (if any). If the invocation failed, `fetchOutputs` raises the exception. This method can only be invoked once.
- `cancel()`
  If the invocation hasn't already completed either successfully or exceptionally, cancelling the future causes it to complete with an instance of `InvocationCanceledException`. Cancellation prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. Cancellation is a local operation and has no effect on the server.
- `ID`
  A numeric value that uniquely identifies the invocation.
- `NumOutputArguments`
  A numeric value denoting how many results the invocation will return upon successful completion.
- `Operation`
  The name of the operation that was invoked.
- `Read`
  A logical value indicating whether the results have already been obtained via `fetchOutputs`.
- `State`
  A string value indicating the current state of the invocation. For a twoway invocation, the property value transitions from `running` to `sent` to `finished`. For a oneway, datagram, or batch invocation, the property value transitions from `running` to `finished`.

Asynchronous Oneway Invocations in MATLAB

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous proxy method on a oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

The future returned for a oneway invocation completes as soon as the request is successfully written to the client-side transport. The future completes exceptionally if an error occurs before the request is successfully written.

Asynchronous Batch Requests in MATLAB

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous proxy method on a batch oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

The future returned for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The future completes exceptionally if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the application until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

The proxy method `ice_flushBatchRequestsAsync` flushes any batch requests queued by that proxy. In addition, similar methods are available on the communicator and the `Connection` object. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

See Also

- Request Contexts
- Batched Invocations

slice2matlab Command-Line Options

The Ice for MATLAB distribution includes a native executable for the Slice-to-MATLAB compiler, `slice2matlab`, as well as a MATLAB script with the same name that you can invoke from the MATLAB console.

The compiler offers the command-line options described below in addition to the standard options.

- `--list-generated`
  Emit a list of generated files in XML format.

See Also

- Using the Slice Compilers

Objective-C Mapping

The Objective-C mapping supports the two memory management models provided with the Objective-C language:

- the "manual release-retain" (MRR) method where the application is responsible for calling `retain` and `release`.
- the "automatic reference counting" (ARC) method where the compiler takes care of inserting the appropriate memory management calls.

The generated code is the same for both models, you don't have to compile your Slice files with different slice2objc compiler options to generate code for one model or the other. The generated code uses preprocessor macros to check if ARC is enabled. Unless specified, the examples shown in the following sections of the mapping use ARC.

Topics

Initialization in Objective-C

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `ICECommunicator` object.

An `ICECommunicator` is a local Objective-C object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `ICECommunicator` object, although it is possible and occasionally desirable to have multiple `ICECommunicator` objects in the same application.

You initialize the Ice run time by calling `createCommunicator` on class `ICEUtil`. `createCommunicator` returns an instance of type `id <ICECommunicator>`:

Objective-C

```
id<ICECommunicator> communicator = [ICEUtil createCommunicator:&argc
argv:argv];
```

`createCommunicator` accepts a *pointer* to `argc` as well as `argv`. The class method scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `createCommunicator` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `createCommunicator` throws an exception.

Before leaving your `main` function, you must call `Communicator::destroy`. The `destroy` method is responsible for finalizing the Ice run time. In particular in a server, `destroy` waits for any operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your `main` function to terminate without calling `destroy` first.

The general shape of our `main` function is therefore:

Objective-C

```
#import <objc/Ice.h>

int
main(int argc, char* argv[])
{
    int status = EXIT_SUCCESS;
    @autoreleasepool
    {
        id<ICECommunicator> communicator = nil;
        @try
        {
            communicator = [ICEUtil createCommunicator:&argc argv:argv];
            ...
        }
        @catch(NSException* ex)
        {
            NSLog(@"%@", ex);
            status = EXIT_FAILURE;
        }

        [communicator destroy];
    }
    return status;
}
```

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-Objective-C Mapping

The client-side Slice-to-Objective-C mapping defines how Slice data types are translated to Objective-C types, and how clients invoke operations, pass parameters, and handle errors. Much of the Objective-C mapping is intuitive. For example, Slice dictionaries map to Cocoa framework dictionaries, so there is little new you have to learn in order to use Slice dictionaries in Objective-C.

The Objective-C mapping is thread-safe. For example, you can concurrently invoke operations on an object from different threads without the risk of race conditions or corrupting data structures in the Ice run time, but you must still synchronize access to application data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the Objective-C mapping, you should need no more than the Slice definition of your application and knowledge of the Objective-C mapping rules. In particular, looking through the generated code in order to discern how to use the Objective-C mapping is likely to be confusing because the generated code is not necessarily meant for human consumption and, occasionally, contains various cryptic constructs to deal with mapping internals. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **ICE, ICEMX, GLACIER2, ICESTORM, ICEGRID Prefixes**
>
> All of the APIs for the Ice run time are prefixed by `ICE`, to avoid clashes with definitions for other libraries or applications. Parts of the Ice API are generated from Slice definitions; other parts provide special-purpose definitions that do not have a corresponding Slice definition. Regardless of the way they are defined, the `ICE` prefix universally applies to all entry points in the Ice run time. We will incrementally cover the contents of the Ice API throughout the remainder of the manual.
>
> The APIs for IceMX, Glacier2, IceStorm and IceGrid are also prefixed with respectively ICEMX, GLACIER2, ICESTORM and ICEGRID.

Topics

Objective-C Mapping for Modules

Because Objective-C does not support namespaces, a Slice module maps to a prefix for the identifiers defined inside the modules. By default, the prefix is the same as the name of the module:

Slice

```
module example
{
    enum Color { Red, Green, Blue }
}
```

With this definition, the Slice identifier `Color` maps to the Objective-C identifier `exampleColor`.

For nested modules, by default, the module identifiers are concatenated. For example, consider the following Slice definition:

Slice

```
module outer
{
    module inner
    {
        enum Color { Red, Green, Blue }
    }
}
```

With this definition, the Slice identifier `Color` becomes `outerinnerColor` in Objective-C.

You can use a metadata directive to change the default mapping to a different prefix. For example:

Slice

```
["objc:prefix:OUT"]
module outer
{
    enum Vehicle { Car, Truck, Bicycle }

    module inner
    {
        enum Color { Red, Green, Blue }
    }
}
```

With this definition, `Vehicle` maps to `OUTVehicle`. However, `Color` still maps to `outerinnerColor`, that is, the metadata directive applies only to types defined in the `outer` module, but not to types that are defined in nested modules. If you want to assign a prefix for types in the nested module, you must use a separate metadata directive, for example:

Slice

```
["objc:prefix:OUT"]
module outer
{
    enum Vehicle { Car, Truck, Bicycle }

    ["objc:prefix:IN"]
    module inner
    {
        enum Color { Red, Green, Blue }
    }
}
```

With this definition, `Vehicle` maps to `OUTVehicle`, and `Color` maps to `INColor`.

For the remainder of the examples in this chapter, we assume that Slice definitions are enclosed by a module `Example` that is annotated with the metadata directive `["objc:prefix:EX"]`.

See Also

- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Structures
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries
- Objective-C Mapping for Constants
- Objective-C Mapping for Exceptions
- Objective-C Mapping for Interfaces

Objective-C Mapping for Identifiers

Objective-C identifiers are derived from Slice identifiers. The exact Objective-C identifier that is generated depends on the context. For types that are nested in modules (and hence have global visibility in Objective-C), the generated Objective-C identifiers are prefixed with their module prefix. Slice identifiers that do not have global visibility (such as operation names and structure members) do not use the module prefix and are preserved without change. For example, consider the following Slice definition:

Slice

```
["objc:prefix:EX"]
module Example
{
    struct Point
    {
        double x;
        double y;
    }
}
```

This maps to the following Objective-C definition:

Objective-C

```
@interface EXPoint : NSObject <NSCopying>
{
    @private
        ICEDouble x;
        ICEDouble y;
}

@property(nonatomic, assign) ICEDouble x;
@property(nonatomic, assign) ICEDouble y;

// More definitions here...
@end
```

If a Slice identifier is the same as an Objective-C keyword, the corresponding Objective-C identifier has an underscore suffix. For example, Slice `while` maps to Objective-C `while_`.

In some cases, the Objective-C mapping generates more than one identifier for a given Slice construct. For example, an interface `Intf` generates the identifiers `EXIntf` and `EXIntfPrx`. If a Slice identifier happens to be an Objective-C keyword, the underscore suffix applies only where necessary, so an interface `while` generates `EXWhile` and `EXWhilePrx`.

Note that Slice operation and member names can clash with the name of an inherited method, property, or instance variable. For example:

Slice

```
exception Failed
{
    string reason; // Clashes with NSException
}
```

This is a legal Slice definition. However, the generated exception class derives from `NSException`, which defines a `reason` method. To avoid hiding the method in the base class, the generated exception class maps the Slice `reason` member to the Objective-C property `reason_`, just as it would for a keyword.

This escape mechanism applies to all generated classes that directly or indirectly derive from `NSObject` or `NSException`.

See Also

- Objective-C Mapping for Modules
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Structures
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries
- Objective-C Mapping for Constants
- Objective-C Mapping for Exceptions
- Objective-C Mapping for Interfaces

Objective-C Mapping for Built-In Types

The Slice built-in types are mapped to Objective-C types as shown below.

Slice	Objective-C
`bool`	`BOOL`
`byte`	`ICEByte`
`short`	`ICEShort`
`int`	`ICEInt`
`long`	`ICELong`
`float`	`ICEFloat`
`double`	`ICEDouble`
`string`	`NSString` or `NSMutableString`

Slice `bool` maps to Objective-C `BOOL`. The remaining integral and floating-point types map to Objective-C type definitions instead of native types. This allows the Ice run time to provide a definition as appropriate for each target architecture. (For example, `ICELong` might be defined as `long` on one architecture and as `long long` on another.)

Note that `ICEByte` is a typedef for `unsigned char`. This guarantees that byte values are always in the range 0..255, and it ensures that right-shifting an `ICEByte` does not cause sign-extension.

Whether a Slice string maps to `NSString` or `NSMutableString` depends on the context. `NSMutableString` is used in some cases for operation parameters; otherwise, if a string is a data member of a Slice structure, class, or exception, it maps to `NSString`. (We will discuss these differences in more detail as we cover the mapping of the relevant Slice language features.)

See Also

- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Structures
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries
- Objective-C Mapping for Constants
- Objective-C Mapping for Exceptions
- Objective-C Mapping for Interfaces

Objective-C Mapping for Enumerations

A Slice enumeration maps to the corresponding enumeration in Objective-C. For example:

Slice

```
["objc:prefix:EX"]
module Example
{
    enum Fruit { Apple, Orange, Pear }
}
```

The generated Objective-C definition is:

Objective-C

```
typedef enum
{
    EXApple, EXOrange, EXPear
} EXFruit;
```

The metadata directive `objc:scoped` allows you to add the enumeration's name as prefix to all enumerators. For example:

Slice

```
["objc:prefix:EX"]
module Example
{
    ["objc:scoped"] enum Fruit { Apple, Orange, Pear }
}
```

The generated Objective-C definition is:

Objective-C

```
typedef enum
{
    EXFruitApple, EXFruitOrange, EXFruitPear
} EXFruit;
```

See Also

- Slice Metadata Directives
- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Structures
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries

Objective-C Mapping for Structures

On this page:

Basic Objective-C Mapping for Structures

A Slice structure maps to an Objective-C class.

For each Slice data member, the generated Objective-C class has a corresponding property. For example, here is our `Employee` structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The Slice-to-Objective-C compiler generates the following definition for this structure:

Objective-C

```
@interface EXEmployee : NSObject <NSCopying>
{
    @private
        ICELong number;
        NSString *firstName;
        NSString *lastName;
}

@property(nonatomic, assign) ICELong number;
@property(nonatomic, strong) NSString *firstName;
@property(nonatomic, strong) NSString *lastName;

-(id) init:(ICELong)number firstName:(NSString *)firstName
lastName:(NSString *)lastName;
+(id) employee:(ICELong)number firstName:(NSString *)firstName lastName
:(NSString *)lastName;
+(id) employee;
// This class also overrides copyWithZone,
// hash, isequal, and dealloc.
@end
```

Mapping for Data Members in Objective-C

For each data member in the Slice definition, the Objective-C class contains a corresponding private instance variable of the same name, as well as a property definition that allows you to set and get the value of the corresponding instance variable. For example, given an instance of `EXEmployee`, you can write the following:

Objective-C

```
ICELong number;
EXemployee *e = ...;
[e setNumber:99];
number = [e number];

// Or, more concisely with dot notation:

e.number = 99;
number = e.number;
```

Properties that represent data members always use the `nonatomic` property attribute. This avoids the overhead of locking each data member during access. The second property attribute is `assign` for integral and floating-point types and `strong` for all other types (such as strings, structures, and so on). If not using automatic reference counting (ARC), the `strong` property attribute is replaced with `retain`.

Note that, for types that have immutable and mutable variants (strings, sequences, and dictionaries), the corresponding data member uses the immutable variant. This allows the application to assign an immutable object to the data member. You can safely cast the data member to the mutable variant if the structure was created by the Ice run time: the unmarshaling code always creates and assigns the mutable version to the data member.

Creating and Initializing Structures in Objective-C

Structures provide the typical (inherited) `init` method:

Objective-C

```
EXEmployee *e = [[EXEmployee alloc] init];
// ...
```

As usual, `init` initializes the instance variables of the structure with zero-filled memory, with the following exceptions:

- A string data member is initialized to an empty string
- An enumerator data member is initialized to the first enumerator
- A structure data member is initialized to a default-constructed value

You can also declare default values in your Slice definition, in which case this `init` method initializes each data member with its declared value instead.

In addition, a structure provides a second `init` method that accepts one parameter for each data member of the structure:

Objective-C

```
-(id) init:(ICELong)number firstName:(NSString *)firstName
lastName:(NSString *)lastName;
```

Note that the first parameter is always unlabeled; the second and subsequent parameters have a label that is the same as the name of the corresponding Slice data member. The additional `init` method allows you to instantiate a structure and initialize its data members in a

single statement:

Objective-C

```
EXEmployee *e = [[EXEmployee alloc] init:99 firstName:@"Brad"
lastName:@"Cox"];
// ...
```

`init` applies the memory management policy of the corresponding properties, that is, it calls `retain` on the `firstName` and `lastName` ar guments.

Each structure also provides two convenience constructors that mirror the `init` methods: a parameter-less convenience constructor and one that has a parameter for each Slice data member:

Objective-C

```
+(id) employee;
+(id) employee:(ICELong)number firstName:(NSString *)firstName
lastName:(NSString *)lastName;
```

The convenience constructors have the same name as the mapped Slice structure (without the module prefix). As usual, they allocate an instance, perform the same initialization actions as the corresponding `init` methods:

Objective-C

```
EXEmployee *e = [EXEmployee employee:99 firstName:@"Brad"
lastName:@"Cox"];
// ...
```

If not using ARC, the convenience constructor calls `autorelease` on the return structure.

Copying Structures in Objective-C

Structures implement the `NSCopying` protocol. In other words, the copy is shallow:

Objective-C

```
EXEmployee *e = [EXEmployee employee:99 firstName:@"Brad"
lastName:@"Cox"];
EXEmployee *e2 = [e copy];
NSAssert(e.number == e2.number);
NSAssert([e.firstName == e2.firstName]); // Same instance
// ...
```

Note that, if you assign an `NSMutableString` to a structure member and use the structure as a dictionary key, you must not modify the string inside the structure without copying it because doing so will corrupt the dictionary.

When not using ARC, structures are copied by assigning instance variables of value type and calling `retain` on each instance variable of non-value type. With ARC, the copied structure shares ownership of instance variables with the original structure.

Deallocating Structures in Objective-C

When not using ARC, each structure implements a `dealloc` method that calls `release` on each instance variable with a `retain` property attribute. This means that structures take care of the memory management of their contents: releasing a structure automatically releases all its instance variables. The `dealloc` method is not defined when ARC is enabled since ARC automatically handles the release of instance variables.

Structure Comparison and Hashing in Objective-C

Structures implement `isEqual`, so you can compare them for equality. Two structures are equal if all their instance variables are equal. For value types, equality is determined by the == operator; for non-value types other than classes, equality is determined by the corresponding instance variable's `isEqual` method. Classes are compared by comparing their identity: two class members are equal if they both point at the same instance.

The `hash` method returns a hash value is that is computed from the hash value of all of the structure's instance variables.

See Also

- Structures
- Dictionaries
- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries
- Objective-C Mapping for Constants
- Objective-C Mapping for Exceptions
- Objective-C Mapping for Interfaces
- Objective-C Mapping for Classes

Objective-C Mapping for Sequences

The Objective-C mapping uses different mappings for sequences of value types (such as `sequence<byte>`) and non-value types (such as `sequence<string>`).

On this page:

Mapping for Sequences of Value Types in Objective-C

The following Slice types are value types:

- Integral types (`bool`, `byte`, `short`, `int`, `long`)
- Floating point types (`float`, `double`)
- Enumerated types

Sequences of these types map to a type definition. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }

sequence<byte> ByteSeq;
sequence<int> IntSeq;
sequence<Fruit> FruitSeq;
```

The three Slice sequences produce the following Objective-C definitions:

Objective-C

```
typedef enum
{
    EXApple, EXPear, EXOrange
} EXFruit;

typedef NSData EXByteSeq;
typedef NSMutableData EXMutableByteSeq;

typedef NSData EXIntSeq;
typedef NSMutableData EXMutableIntSeq;

typedef NSData EXFruitSeq;
typedef NSMutableData EXMutableFruitSeq;
```

As you can see, each sequence definition creates a pair of type definitions, an immutable version named <*`module-prefix`*><*`Slice-name`*>, and a mutable version named <*`module-prefix`*>`Mutable`<*`Slice-name`*>. This constitutes the entire public API for sequences of value types, that is, sequences of value types simply map to `NSData` or `NSMutableData`. The `NS(Mutable)Data` sequences contain an array of the corresponding element type in their internal byte array.

> We chose to map sequences of value types to `NSData` instead of `NSArray` because of the large overhead of placing each sequence element into an `NSNumber` container instance.

For example, here is how you could initialize a byte sequence of 1024 elements with values that are the modulo 128 of the element index in reverse order:

Objective-C

```
int limit = 1024;
EXMutableByteSeq *bs = [NSMutableData dataWithLength:limit];
ICEByte *p = (ICEByte *)[bs bytes];
while(--limit > 0)
{
    *p++ = limit % 0x80;
}
```

Naturally, you do not need to initialize the sequence using a loop. For example, if the data is available in a buffer, you could use the `dataWithBytes:length` or `dataWithBytesNoCopy:length` methods of `NSData` instead.

Here is one way to retrieve the bytes of the sequence:

Objective-C

```
const ICEByte* p = (const ICEByte *)[bs bytes];
const ICEByte* limitp = p + [bs length];
while(p < limitp)
{
    printf("%d\n", *p++);
}
```

For sequences of types other than `byte` or `bool`, you must keep in mind that the length of the `NSData` array is not the same as the number of elements. The following example initializes an integer sequence with the first few primes and prints out the contents of the sequence:

Objective-C

```
const int primes[] = { 1, 2, 3, 5, 7, 9, 11, 13, 17, 19, 23 };
EXMutableIntSeq *is = [NSMutableData dataWithBytes:primes
length:sizeof(primes)];

const ICEInt *p = (const ICEInt *)[is bytes];
int limit = [is length] / sizeof(*p);
int i;
for(i = 0; i < limit; ++i)
{
    printf("%d\n", p[i]);
}
```

The code to manipulate a sequence of enumerators is very similar. For portability, you should not assume a particular size for enumerators. That is, instead of relying on all enumerators having the size of, for example, an `int`, it is better to use `sizeof(EXFruit)` to ensure that you are not overstepping the bounds of the sequence.

Mapping of Sequences of Non-Value Types in Objective-C

Sequences of non-value types, such as sequences of `string`, structures, classes, and so on, map to mutable and immutable type

definitions of `NSArray`. For example:

Slice

```
sequence<string> Page;
sequence<Page> Book;
```

This maps to:

Objective-C

```
typedef NSArray EXPage;
typedef NSMutableArray EXMutablePage;

typedef NSArray EXBook;
typedef NSMutableArray EXMutableBook;
```

You use such sequences as you would use any other `NSArray` in your code. For example:

Objective-C

```
EXMutablePage *page1 = [NSArray arrayWithObjects:
                                @"First line of page one",
                                @"Second line of page one",
                                nil];

EXMutablePage *page2 = [NSArray arrayWithObjects:
                                @"First line of page two",
                                @"Second line of page two",
                                nil];

EXMutableBook *book = [NSMutableArray array];
[book addObject:page1];
[book addObject:page2];
[book addObject:[NSArray array]]; // Empty page
```

This creates a book with three pages; the first two pages contain two lines each, and the third page is empty. You can print the contents of the book as follows:

Objective-C

```
int pageNum = 0;
for(EXPage *page in book)
{
    ++pageNum;
    int lineNum = 0;
    if([page count] == 0)
    {
        printf("page %d: <empty>\n", pageNum);
    }
    else
    {
        for(NSString *line in page)
        {
            ++lineNum;
            printf("page %d, line %d: %s\n",
pageNum, lineNum, [line UTF8String]);
        }
    }
}
```

This prints:

```
page 1, line 1: First line of page one
page 1, line 2: Second line of page one
page 2, line 1: First line of page two
page 2, line 2: Second line of page two
page 3: <empty>
```

If you have a sequence of proxies or a sequence of classes, to transmit a null proxy or class inside a sequence, you must insert an `NSNull` value into the `NSArray`. In addition, the mapping also allows you to use `NSNull` as the element value of an `NSArray` for elements of type string, structure, sequence, or dictionary. For example, instead of inserting an empty `NSArray` into the book sequence in the preceding example, we could also have inserted `NSNull`:

Objective-C

```
EXMutableBook *book = [NSMutableArray array];
[book addObject:page1];
[book addObject:page2];
[book addObject:[NSNull null]]; // Empty page
```

See Also

- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Structures
- Objective-C Mapping for Dictionaries

Objective-C Mapping for Dictionaries

Here is the definition of our `EmployeeMap` once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

The following code is generated for this definition:

Objective-C

```
typedef NSDictionary EXEmployeeMap;
typedef NSMutableDictionary EXMutableEmployeeMap;
```

Similar to sequences, Slice dictionaries map to type definitions for `NSDictionary` and `NSMutableDictionary`, with the names <*module-prefix*><*Slice-name*> and <*module-prefix*>Mutable<*Slice-name*>.

As a result, you can use the dictionary like any other `NSDictionary`, for example:

Objective-C

```
EXMutableEmployeeMap *em = [EXMutableEmployeeMap dictionary];
EXEmployee *e = [EXEmployee employee];
e.number = 42;
e.firstName = @"Stan";
e.lastName = @"Lippman";
[em setObject:e forKey:[NSNumber numberWithLong:e.number]];

e = [EXEmployee employee];
e.number = 77;
e.firstName = @"Herb";
e.lastName = @"Sutter";
[em setObject:e forKey:[NSNumber numberWithLong:e.number]];
```

To put a value type into a dictionary (either as the key or the value), you must use `NSNumber` as the object to hold the value. If you have a dictionary that uses a Slice enumeration as the key or the value, you must insert the enumerator as an `NSNumber` that holds an `int`.

To insert a null proxy or null class instance into a dictionary as a value, you must insert `NSNull`.

As a convenience feature, the Objective-C mapping also allows you to insert `NSNull` as the value of a dictionary if the value type of the dictionary is a string, structure, sequence, or dictionary. If you send such a dictionary to a receiver, the Ice run time marshals an empty string, default-initialized structure, empty sequence, or empty dictionary as the corresponding value to the receiver, respectively.

See Also

- Dictionaries
- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Structures

Objective-C Mapping for Constants

Slice constant definitions map to corresponding Objective-C constant definitions. For example:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

Objective-C

```
static const BOOL EXAppendByDefault = YES;
static const ICEByte EXLowerNibble = 15;
static NSString * const EXAdvice = @"Don't Panic!";
static const ICEShort EXTheAnswer = 42;
static const ICEDouble EXPI = 3.1416;

typedef enum
{
    EXApple, EXPear, EXOrange
} EXFruit;
static const EXFruit EXFavoriteFruit = EXPear;
```

All constants are initialized directly in the generated header file, so they are compile-time constants and can be used in contexts where a compile-time constant expression is required, such as to dimension an array or as the `case` label of a `switch` statement.

Slice string literals that contain non-ASCII characters or universal character names are mapped to Objective-C string literals with these characters replaced by their UTF-8 encoding as octal escapes. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

Objective-C

```
static NSString * const demoEgg = @"\305\223uf";
static NSString * const demoHeart = @"c\305\223ur";
static NSString * const demoBanana = @"\360\237\215\214";
```

See Also

- Constants and Literals
- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Structures
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries
- Objective-C Mapping for Exceptions
- Objective-C Mapping for Interfaces

Objective-C Mapping for Exceptions

This page describes the Objective-C mapping for exceptions.

On this page:

Exception Inheritance Hierarchy in Objective-C

Here again is a fragment of the Slice definition for our world time server:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

Objective-C

```
@interface EXGenericError : ICEUserException
{
    NSString *reason_;
}

@property(nonatomic, strong) NSString *reason_;

// ...
@end

@interface EXBadTimeVal : EXGenericError
// ...
@end

@interface EXBadZoneName : EXGenericError
// ...
@end
```

Each Slice exception is mapped to an Objective-C class. For each exception member, the corresponding class contains a private instance variable and a property. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

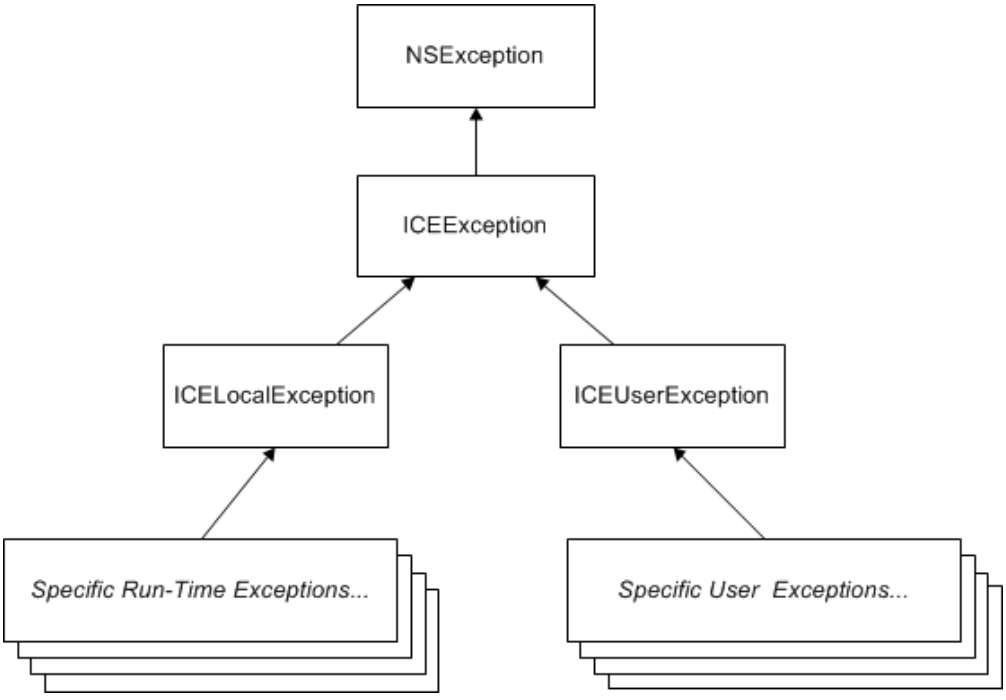
The inheritance structure of the Slice exceptions is preserved for the generated classes, so `EXBadTimeVal` and `EXBadZoneName` inherit from `EXGenericError`.

In turn, `EXGenericError` derives from `ICEUserException`:

Objective-C

```
@interface ICEException : NSException
-(NSString*)ice_id;
@end

@interface ICEUserException : ICEException
// ...
@end

@interface ICELocalException : ICEException
// ...
@end
```

Note that `ICEUserException` itself derives from `ICEException`, which derives from `NSException`. Similarly, run-time exceptions derive from a common base class `ICELocalException` that derives from `ICEException`, so we have the inheritance structure shown below:



Inheritance structure for exceptions.

`ICEException` provides a single method, `ice_id`, that returns the Slice type ID of the exception. For example, the return value of `ice_id` for our Slice `GenericError` in module `M` is `::M::GenericError`.

Mapping for Exception Data Members in Objective-C

As we mentioned earlier, each data member of a Slice exception generates a corresponding Objective-C property. Here is an example that extends our `GenericError` with yet another exception:

Slice

```
exception GenericError
{
    string reason;
}

exception FileError extends GenericError
{
    string name;
    int errorCode;
}
```

The generated properties for these exceptions are as follows:

Objective-C

```
@interface EXGenericError : ICEUserException
{
    NSString *reason_;
}

@property(nonatomic, strong) NSString *reason_;

// ...
@end

@interface EXFileError : EXGenericError
{
    NSString *name_;
    ICEInt errorCode;
}

@property(nonatomic, strong) NSString *name_;
@property(nonatomic, assign) ICEInt errorCode;

// ...
@end
```

This is exactly the same mapping as for structure members, with one difference: the `name` and `reason` members map to `name_` and `reason_` properties, whereas — as for structures — `errorCode` maps to `errorCode`. The trailing underscore for `reason_` and `name_` prevents a name collision with the `name` and `reason` methods that are defined by `NSException`: if you call the `name` method, you receive the name that is stored by `NSException`; if you call the `name_` method, you receive the value of the `name_` instance variable of `EXFileError`:

Objective-C

```
@try
{
    // Do something that can throw ExFileError...
}
@catch(EXFileError *ex)
{
    // Print the value of the Slice reason, name,
    // and errorCode members.
    printf("reason: %s, name: %s, errorCode: %d\n",
           [ex.reason_ UTF8String],
           [ex.name_ UTF8String],
           ex.errorCode);

    // Print the NSException name.
    printf("NSException name: %s\n", [[ex name] UTF8String]);
}
```

The same escape mechanism applies if you define exception data members named `callStackReturnAddresses`, `raise`, or `userInfo`.

Objective-C Mapping for User Exceptions

Initialization of exceptions follows the same pattern as for structures: each exception (apart from the inherited no-argument `init` method) provides an `init` method that accepts one argument for each data member of the exception, and two convenience constructors. For example, the generated methods for our `EXGenericError` exception look as follows:

Objective-C

```
@interface EXGenericError : ICEUserException
// ...

-(id) init:(NSString *)reason;
+(id) genericError;
+(id) genericError:(NSString *)reason;
@endif
```

If a user exception has no data members (and its base exceptions do not have data members either), only the inherited `init` method and the no-argument convenience constructor are generated.

The no-argument `init` method and the no-argument convenience constructor initialize the instance variables of the exception with zero-filled memory, with the following exceptions:

- A string data member is initialized to an empty string
- An enumerator data member is initialized to the first enumerator
- A structure data member is initialized to a default-constructed value

If you declare default values in your Slice definition, the inherited `init` method and the no-argument convenience constructor initialize each data member with its declared value instead.

If an exception has a base exception with data members, its `init` method and convenience constructor accept one argument for each Slice

data member, in base-to-derived order. For example, here are the methods for the `FileError` exception we defined above:

Objective-C

```
@interface EXFileError : EXGenericError
// ...

-(id) init:(NSString *)reason name_:(NSString *)name
errorCode:(ICEInt)errorCode;
+(id) fileError;
+(id) fileError:(NSString *)reason name_:(NSString *)name
errorCode:(ICEInt)errorCode;
@end
```

Note that `init` and the second convenience constructor accept three arguments; the first initializes the `EXGenericError` base, and the remaining two initialize the instance variables of `EXFileError`.

Objective-C Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `ICELocalException` which, in turn, derives from `ICEException`. (See the above illustration for an example of an inheritance diagram.)

By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `NSException`
  This is the root of the complete inheritance tree. Catching `NSException` catches all exceptions, whether they relate to Ice or the Cocoa framework.
- `ICEException`
  Catching `ICEException` catches both user and run-time exceptions.
- `ICEUserException`
  This is the root exception for all user exceptions. Catching `ICEUserException` catches all user exceptions (but not run-time exceptions).
- `ICELocalException`
  This is the root exception for all run-time exceptions. Catching `ICELocalException` catches all run-time exceptions (but not user exceptions).
- `ICETimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `ICEConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, an `ICEConnectTimeoutException` can be handled as `ICEConnectTimeoutException`, `ICETimeoutException`, `ICELocalException`, `ICEException`, or `NSException`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `ICELocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `ICEFacetNotExistException` and `ICEObjectNotExistException`, respectively.

Creating and Initializing Run-Time Exceptions in Objective-C

`ICELocalException` provides two properties that return the file name and line number at which an exception was raised:

Objective-C

```
@interface ICELocalException : ICEException
// ...
@property(nonatomic, readonly) NSString* file;
@property(nonatomic, readonly) int line;

-(id)init:(const char*)file line:(int)line;
+(id)localException:(const char*)file line:(int)line;
@end
```

The `init` method and the convenience constructor accept the file name and line number as arguments.

Concrete run-time exceptions that derived from `ICEException` provide a corresponding `init` method and convenience constructor. For example, here is the Slice definition of `ObjectNotExistException`:

Slice

```
local exception RequestFailedException
{
    Identity id;
    string facet;
    string operation;
}

local exception ObjectNotExistException
extends RequestFailedException {}
```

The Objective-C mapping for `ObjectNotExistException` is:

Objective-C

```
@interface ICEObjectNotExistException : ICERequestFailedException
// ...
-(id) init:(const char*)file__p line:(int)line__p;
-(id) init:(const char*)file__p
          line:(int)line__p
          id_:(ICEIdentity *)id_
          facet:(NSString *)facet
          operation:(NSString *)operation;
+(id) objectNotExistException:(const char*)file__p
                             line:(int)line__p;
+(id) objectNotExistException:(const char*)file__p
                             line:(int)line__p
                             id_:(ICEIdentity *)id_
                             facet:(NSString *)facet
                             operation:(NSString *)operation;
@end
```

In other words, as for user exceptions, run-time exceptions provide `init` methods and convenience constructors that accept arguments in base-to-derived order. This means that all run-time exceptions require a file name and line number when they are instantiated. For example, you can throw an `ICEObjectNotExistException` as follows:

Objective-C

```
@throw [ICEObjectNotExistException
objectNotExistException:__FILE__ line:__LINE__];
```

If you throw this exception in the context of an executing operation on the server side, the `id_`, `facet`, and `operation` instance variables are automatically initialized by the Ice run time.

When you instantiate a run-time exception, the base `NSException` is initialized such that its `name` method returns the same string as `ice_name`; the `reason` and `userInfo` methods return `nil`.

Copying and Deallocating Exceptions in Objective-C

User exceptions and run-time exceptions implement the `NSCopying` protocol, so you can copy them. The semantics are the same as for structures.

Similarly, like structures, when not using ARC, exceptions implement a `dealloc` method that takes care of deallocating the instance variables when an exception is released.

Exception Comparison and Hashing in Objective-C

Exceptions do not override `isEqual` or `hash`, so these methods have the behavior inherited from `NSObject`.

See Also

- User Exceptions
- Run-Time Exceptions
- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Structures
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries
- Objective-C Mapping for Constants
- Objective-C Mapping for Interfaces
- Versioning
- Object Life Cycle

Objective-C Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that represents the remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

On this page:

Proxy Classes and Proxy Protocols in Objective-C

On the client side, interfaces map to a protocol with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

Slice

```
["objc:prefix:EX"]
module Example
{
    interface Simple
    {
        void op();
    }
}
```

The Slice compiler generates the following definitions for use by the client:

Objective-C

```
@interface EXSimplePrx : ICEObjectPrx
// Mapping-internal methods here...
@end

@protocol EXSimplePrx <ICEObjectPrx>
-(void) op;
-(void) op:(ICEContext *)context;
@end;
```

As you can see, the compiler generates a proxy protocol `EXSimplePrx` and a proxy class `EXSimplePrx`. In general, the generated name for both protocol and class is <*`module-prefix`*><*`interface-name`*>`Prx`.

In the client's address space, an instance of `EXSimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `EXSimplePrx` derives from `ICEObjectPrx`, and that `EXSimplePrx` adopts the `ICEObjectPrx` protocol. This reflects the fact that all Slice interfaces implicitly derive from `Ice::Object`. For each operation in the interface, the proxy protocol has two methods whose name is derived from the operation. For the preceding example, we find that the operation `op` is mapped to two methods, `op` and `op:`.

The second method has a trailing parameter of type `ICEContext`. This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist. (We examine the `ICEContext` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Interface Inheritance in Objective-C

Inheritance relationships among Slice interfaces are maintained in the generated Objective-C code. For example:

Slice

```
["objc:prefix:EX"]
module Example
{
    interface A { ... }
    interface B { ... }
    interface C extends A, B { ... }
}
```

The generated code reflects the inheritance hierarchy:

Objective-C

```
@interface EXCPrx : ICEObjectPrx <EXCPrx>
    ...
@end

@protocol EXCPrx <EXAPrx, EXBPrx>
@end
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

Proxy Instantiation and Casting in Objective-C

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

Proxies are immutable: once the run time has instantiated a proxy, that proxy continues to denote the same remote object and cannot be changed. `ICEObjectPrx` implements `NSCopying`. However, calling `copy` returns a reference on the target proxy.

Proxies are always passed and returned as type `id<<`*`module-prefix`*`><`*`interface-name`*`>Prx>`. For example, for the preceding `Simple` interface, the proxy type is `id<EXSimplePrx>`.

The `ICEObjectPrx` base class provides class methods that allow you to cast a proxy from one type to another, as described below.

Using a Checked Cast in Objective-C

A `checkedCast` tests whether the object denoted by a proxy implements the specified interface:

Objective-C

```
+(id) checkedCast:(id<ICEObjectPrx>)proxy;
```

If so, the cast returns a proxy to the specified interface; otherwise, if the object denoted by the proxy does not implement the specified interface, the cast returns `nil`. Checked casts are typically used to safely down-cast a proxy to a more derived interface. For example, assuming we have Slice interfaces `Base` and `Derived`, you can write the following:

Objective-C

```
id<EXBasePrx> base = ...;  // Initialize base proxy
id<EXDerivedPrx> derived = [EXDerivedPrx checkedCast:base];
if(derived != nil)
{
    // base implements run-time type Derived
    // use derived...
}
else
{
    // Base has some other, unrelated type
}
```

The expression [`EXDerivedPrx checkedCast:base`] tests whether `base` points at an object of type `Derived` (or an object with a type that is derived from `Derived`). If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to `nil`. (Proxies that "point nowhere" are represented by `nil`.)

> Calling `checkedCast` on a proxy that is already of the desired proxy type returns immediately that proxy. Otherwise, `checkedCast` always calls `ice_isA` on the target object, and upon success, creates a new instance of the desired proxy class.

The message effectively asks the server "is the object denoted by this proxy of type `Derived`?" The reply from the server is communicated to the application code in form of a successful (non-`nil`) or unsuccessful (`nil`) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of a proxy is without confirmation from the server. (For example, the server may replace the implementation of the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive an `ICEConnectionRefusedException`; if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ICEObjectNotExistException`.

Using an Unchecked Cast in Objective-C

In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

Objective-C

```
+(id) uncheckedCast:(id<ICEObjectPrx>)proxy;
```

Here is an example:

Objective-C

```
id<EXBasePrx> base;
base = ...;  // Initialize base to point at a Derived
id<EXDerivedPrx> derived = [EXDerivedPrx uncheckedCast:base];
// Use derived...
```

An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object. You should use an `uncheckedCast` only if you are certain that the proxy indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not

checked in any way; it does not contact the object in the server and, if the proxy does not support the specified interface, the cast does not return null. If you use the proxy resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `ICEOperationNotExistException`, but, depending on the circumstances, the Ice run time may also report an exception indicating that unmarshaling has failed, or even silently return garbage results.

Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization, it is common to receive a proxy of type `id<ICEObjectPrx>`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

Note that an `uncheckedCast` is *not* the same as an ordinary cast. The following is incorrect and has undefined behavior:

Objective-C

```
id<EXDerivedPrx> derived = (id<EXDerivedPrx>)base; // Wrong!
```

When not using ARC, both `checkedCast` and `uncheckedCast` call `autorelease` on the proxy they return so, if you want to prevent the proxy from being deallocated once the enclosing autorelease pool is drained, you must call `retain` on the returned proxy.

Using Proxy Methods in Objective-C

The `ICEObjectPrx` provides a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten-second invocation timeout as shown below:

Objective-C

```
id<ICEObjectPrx> proxy = [communicator stringToProxy:...];
proxy = [proxy ice_invocationTimeout:10000];
```

A factory method returns a new (autoreleased) proxy object if the requested modification differs from the current proxy, otherwise it returns the original proxy. The returned proxy is always of the same type as the source proxy, except for the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in Objective-C

Proxy objects support comparison with `isEqual`. Note that `isEqual` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same as well. In other words, comparison with `isEqual` tests for proxy identity, not object identity. A common mistake is to write code along the following lines:

Objective-C

```
id<ICEObjectPrx> p1 = ...;      // Get a proxy...
id<ICEObjectPrx> p2 = ...;      // Get another proxy...

if(![p1 isEqual:p2])
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen if, for example, `p1` and `p2` embed the same object identity, but use a different protocol to contact the target object. Similarly, the protocols might be the same, but could denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `isEqual`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `isEqual`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use additional methods provided by proxies:

Objective-C

```
@protocol ICEObjectPrx <NSObject, NSCopying>
// ...
-(NSComparisonResult) compareIdentity:(id<ICEObjectPrx>)p;
-(NSComparisonResult) compareIdentityAndFacet:(id<ICEObjectPrx>)p;
@end
```

The `compareIdentity` method compares the object identities embedded in two proxies while ignoring other information, such as facet and transport information. To include the facet name in the comparison, use `compareIdentityAndFacet` instead.

`compareIdentity` and `compareIdentityAndFacet` allow you to correctly compare proxies for object identity. The example below demonstrates how to use `compareIdentity`:

Objective-C

```
id<ICEObjectPrx> p1 = ...;      // Get a proxy...
id<ICEObjectPrx> p2 = ...;      // Get another proxy...

if([p1 compareIdentity:p2] != NSOrderedSame)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Objective-C Mapping for Operations
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

Objective-C Mapping for Operations

On this page:

Basic Objective-C Mapping for Operations

As we saw in the mapping for interfaces, for each operation on an interface, the proxy protocol contains two corresponding methods with the same name as the operation.

To invoke an operation, you call it via the proxy object. For example, here is part of the definitions for our file system:

Slice

```
["objc:prefix:FS"]
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The proxy protocol for the `Node` interface looks as follows:

Objective-C

```
@protocol FSNodePrx <ICEObjectPrx>
-(NSMutableString *) name;
-(NSMutableString *) name:(ICEContext *)context;
@end;
```

The `name` method returns a value of type `NSMutableString`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

Objective-C

```
id<EXNodePrx> node = ...;       // Initialize proxy
NSString *name = [node name];   // Get name via RPC
```

The `name` method sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

It is safe to ignore the return value even when not using ARC as the returned value is autoreleased. For example, the following code contains no memory leak:

Objective-C

```
id<EXNodePrx> node = ...;       // Initialize proxy
[node name];                    // Useless, but no leak
```

If you ignore the return value, no memory leak occurs because the next time the enclosing autorelease pool is drained, the memory will be reclaimed.

Normal and `idempotent` Operations in Objective-C

You can add an `idempotent` qualifier to a Slice operation. As far as the corresponding proxy protocol methods are concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Ops
{
                string op1();
    idempotent  string op2();
    idempotent  void op3(string s);
}
```

The proxy protocol for this interface looks like this:

Objective-C

```
@protocol EXOpsPrx <ICEObjectPrx>
-(NSMutableString *) op1;
-(NSMutableString *) op1:(ICEContext *)context;
-(NSMutableString *) op2;
-(NSMutableString *) op2:(ICEContext *)context;
-(void) op3:(NSString *)s;
-(void) op3:(NSString *)s context:(ICEContext *)context;
@end;
```

For brevity, we will not show the methods with the additional trailing `context` parameter for the remainder of this discussion. Of course, the compiler generates the additional methods regardless.

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the `idempotent` keyword.

Passing Parameters in Objective-C

In-Parameters in Objective-C

The parameter passing rules for the Objective-C mapping are very simple: value type parameters are passed by value and non-value type parameters are passed by pointer. Semantically, the two ways of passing parameters are identical: the Ice run time guarantees not to change the value of an in-parameter.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following code for this definition:

Objective-C

```
@interface EXNumberAndString : NSObject <NSCopying>
// ...
@property(nonatomic, assign) ICEInt x;
@property(nonatomic, strong) NSString *str;
// ...
@end

typedef NSArray EXStringSeq;
typedef NSMutableArray EXMutableStringSeq;

typedef NSDictionary EXStringTable;
typedef NSMutableDictionary EXMutableStringTable;

@protocol EXClientToServerPrx <ICEObjectPrx>
-(void) op1:(ICEInt)i f:(ICEFloat)f b:(BOOL)b s:(NSString *)s;
-(void) op2:(EXNumberAndString *)ns ss:(EXStringSeq *)ss
st:(NSDictionary *)st;
-(void) op3:(id<EXClientToServerPrx>)proxy;
@end;
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

Objective-C

```
id<EXClientToServerPrx> p = ...;           // Get proxy...

[p op1:42 f:3.14 b:YES s:@"Hello world!"]; // Pass literals

ICEInt i = 42;
ICEFloat f = 3.14;
BOOL b = YES;
NSString *s = @"Hello world!";

[p op1:i f:f b:b s:s];                     // Pass simple vars

EXNumberAndString *ns = [EXNumberAndString numberAndString:42
str:@"The Answer"];
EXMutableStringSeq *ss = [ExMutableStringSeq array];
[ss addObject:@"Hello world!"];
EXStringTable *st = [EXMutableStringTable dictionary];
[ss setObject:ss forKey:[NSNumber numberWithInt:0]];

[p op2:ns ss:ss st:st];                    // Pass complex vars

[p op3:p];                                 // Pass proxy
```

You can pass either literals or variables to the various operations. The Ice run time simply marshals the value of the parameters to the server and leaves parameters otherwise untouched, so there are no memory-management issues to consider.

Note that the invocation of `op3` is somewhat unusual: the caller passes the proxy it uses to invoke the operation to the operation as a parameter. While unusual, this is legal (and no memory management issues arise from doing this.)

Passing `nil` and `NSNull` in Objective-C

The Slice language supports the concept of null ("points nowhere") for only two of its types: proxies and classes. For either type, `nil` represents a null proxy or class. For other Slice types, such as strings, the concept of a null string simply does not apply. (There is no such thing as a null string, only the empty string.) However, strings, structures, sequences, and dictionaries are all passed by pointer, which raises the question of how the Objective-C mapping deals with `nil` values.

As a convenience feature, the Objective-C mapping permits passing of `nil` as a parameter for the following types:

- Proxies (`nil` sends a null proxy.)
- Classes (`nil` sends a null class instance.)
- Strings (`nil` sends an empty string.)
- Structures (`nil` sends a default-initialized structure.)
- Sequences (`nil` sends an empty sequence.)
- Dictionaries (`nil` sends an empty dictionary.)

It is impossible to add `nil` to an `NSArray` or `NSDictionary`, so the mapping follows the usual convention that an `NSArray` element or `NSDictionary` value that is conceptually `nil` is represented by `NSNull`. For example, to send a sequence of proxies, some of which are null proxies, you must insert `NSNull` values into the sequence.

As a convenience feature, if you have a sequence with elements of type string, structure, sequence, or dictionary, you can use `NSNull` as the element value. For elements that are `NSNull`, the Ice run time marshals an empty string, default-initialized structure, empty sequence, or empty dictionary to the receiver.

Similarly, for dictionaries with value type string, structure, sequence, or dictionary, you can use `NSNull` as the value to send the corresponding empty value (or default-initialized value, in the case of structures).

Out-Parameters in Objective-C

The Objective-C mapping passes out-parameters by pointer (for value types) and by pointer-to-pointer (for non-value types). Here is the Slice definition once more, modified to pass all parameters in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
out StringSeq ss, out StringTable st);
    void op3(out ClientToServer* proxy);
}
```

The Slice compiler generates the following code for this definition:

Objective-C

```
@protocol EXServerToClientPrx <ICEObjectPrx>
-(void) op1:(ICEInt *)i f:(ICEFloat *)f b:(BOOL *)b
s:(NSMutableString **)s;
-(void) op2:(EXNumberAndString **)ns ss:(EXMutableStringSeq **)ss
st:(EXMutableStringTable **)st;
-(void) op3:(id<EXClientToServerPrx> *)proxy;
@end
```

Note that, for types that come in immutable and mutable variants (strings, sequences, and dictionaries), the corresponding out-parameter uses the mutable variant.

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

Objective-C

```
id<EXServerToClientPrx> p = ...; // Get proxy...

ICEInt i;
ICEFloat f;
BOOL b;
NSMutableString *s;

[p op1:&i f:&f b:&b s:&s];
// i, f, b, and s contain updated values now

EXNumberAndString *ns;
EXStringSeq *ss;
EXStringTable *st;

[p op2:&ns ss:&ss st:&st];
// ns, ss, and st contain updated values now

[p op3:&p];
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes pointers to pointer variables to a method; once the operation completes, the values of those variables will have been set by the server.

Memory Management for Out-Parameters in Objective-C

When the Ice run time returns an out-parameter to the caller, it does not make any assumptions about the previous value of that parameter (if any). In other words, if you pass an initialized string as an out-parameter, the value you pass is simply discarded and the corresponding variable is assigned a new instance. As an example, consider the following operation:

Slice

```
void getString(out string s);
```

You could call this as follows:

Objective-C

```
NSMutableString *s = @"Hello";
[p getString:&s];
// s now points at the returned string.
```

When not using ARC, all out-parameters are autoreleased by the Ice run time before they are returned. With ARC, out parameters are implicitly qualified with `__autoreleasing` so the returned objects are automatically autoreleased. This is convenient because it does just the right thing with respect to memory management. For example, the following code does not leak memory:

Objective-C

```
NSMutableString *s = @"Hello";
[p getString:&s];
[p getString:&s]; // No leak here.
```

Beware however that when not using ARC, because the pointer value of out-parameters is simply assigned by the proxy method, you must be careful not to pass a variable as an out-parameter if that variable was not released or autoreleased:

Objective-C

```
NSMutableString *s = [[NSMutableString alloc] initWithString:@"Hello"];
[p getString:&s]; // Bad news when not using ARC!
```

This code leaks the initial string because the proxy method assigns the passed pointer without calling `release` on it first. (In practice, this is rarely a problem because there is no need to initialize out-parameters and, if an out-parameter was initialized by being passed as an out-parameter to an operation earlier, its value will have been autoreleased by the proxy method already.) When using ARC, this isn't an issue, the compiler will automatically release the string when the out-parameter is returned.

It is worth having another look at the final call of the code example we saw earlier:

Objective-C

```
[p op3:&p];
```

Here, p is the proxy that is used to dispatch the call. That same variable p is also passed as an out-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe (with the caveat we just discussed when not using ARC).

Receiving Return Values in Objective-C

The Objective-C mapping returns return values in much the same way as out-parameters: value types are returned by value, and non-value types are returned by pointer. As an example, consider the following operations:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

interface Ops
{
    int getInt();
    string getString();
    NumberAndString getNumberAndString();
}
```

The proxy protocol looks as follows:

Objective-C

```
@protocol EXOpsPrx <ICEObjectPrx>
-(ICEInt) getInt;
-(NSMutableString *) getString;
-(EXNumberAndString *) getNumberAndString;
@end
```

Note that, for types with mutable and immutable variants (strings, sequences, and dictionaries), the formal return type is the mutable variant. As for out-parameters, when not using ARC, anything returned by pointer is autoreleased by the Ice run time. This means that the following code works fine and does not contain memory management errors whether or not you use ARC:

Objective-C

```
EXNumberAndString *ns = [NSNumberAndString numberAndString];
ns.x = [p getInt];
ns.str = [p getString]; // Autoreleased by getString and retained by
ns.str when not using ARC
[p getNumberAndString]; // No leak here.
```

The return value of `getString` is autoreleased by the proxy method but, during the assignment to the property `str`, the generated code calls `retain`, so the structure keeps the returned string alive in memory, as it should. Similarly, ignoring the return value from an invocation is safe because the returned value is autoreleased and will be reclaimed when the enclosing autorelease pool is drained.

Chained Invocations in Objective-C

Consider the following simple interface containing two operations, one to set a value and one to get it:

Slice

```
interface Name
{
    string getName();
    void setName(string name);
}
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

Objective-C

```
[p2 setName:[p1 getName]]; // No leak here.
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`. There are no memory-management or exception safety issues with this code.

`nil` Out-Parameters and Return Values in Objective-C

If an out-parameter or return value is a proxy or class, and the operation returns a null proxy or class, the proxy method returns `nil`. If a proxy or class is returned as part of a sequence or dictionary, the corresponding entry is `NSNull`.

For strings, structures, sequences, and dictionaries, the Ice run time *never* returns `nil` or `NSNull` (even if the server passed `nil` or `NSNull` as the value). Instead, the unmarshaling code always instantiates an empty string, empty sequence, or empty dictionary, and it always initializes structure members during unmarshaling, so structures that are returned from an operation invocation never contain a `nil` instance variable (except for proxy and class instance variables).

Optional Parameters in Objective-C

The Objective-C mapping uses the `id` type for optional parameters. As a result, there's no compile time check for optional parameters. Instead, Ice performs a run-time type check and if the optional parameter does not match the expected type an `NSException` with the `NSInvalidArgumentException` name is raised. Slice types that map to an Objective-C class use the same mapping as required parameters. Slice built-in basic types (except string) are boxed into an `NSNumber` value. The `ICENone` singleton value can also be passed as the value of an optional parameter or return value.

Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string aString, optional(3) int
anInt, out optional(4) float outFloat);
```

A client can invoke this operation as shown below:

Objective-C

```
id f;
id i;

i = [proxy execute:@"--file log.txt" anInt:@14 outFloat:&f]
i = [proxy execute:ICENone anInt:@14 outFloat:&f] // aString is unset

if(i == ICENone)
{
    NSLog(@"return value is not set");
}
else
{
    int v = [i intValue];
    NSLog(@"return value is set to %d", v);
}
```

Passing `nil` for an optional parameter is the same as passing `nil` for a required parameter, the optional parameter is considered to be set to `nil`. For Slice built-in basic types (except string), the optional parameter is considered to be set to `0` or `NO` for booleans.

A well-behaved program must not assume that an optional parameter always has a value. It should compare the value to `ICENone` to determine whether the optional parameter is set.

Exception Handling in Objective-C

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as Objective-C exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

Objective-C

```
id<EXChildPrx> child = ...;    // Get proxy...
@try
{
    [child askToCleanUp];      // Give it a try...
}
@catch(EXTantrum *t)
{
    printf("The child says: %s\n", t.reason_);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

Objective-C

```
int
main(int argc, char* argv[])
{
    int status = 1;
    @try
    {
       id<EXChildPrx> child = ...;       // Get proxy...
       @try
       {
           [child askToCleanUp];         // Give it a try...
           [child praise];               // Give positive feedback...
       }
       @catch(EXTantrum *t)
       {
           printf("The child says: %s\n", t.reason);
           [child scold];                // Recover from error...
       }
       status = 0;
    }
    @catch(ICELocalException *e)
    {
        printf("Unexpected run-time error: %s\n", [e ice_name]);
    }
    // ...
    return status;
}
```

Exceptions and Out-Parameters in Objective-C

If an operation throws an exception, the Ice run time makes no guarantee for the value of out-parameters. Individual out-parameters may have the old value, the new value, or a value that is indeterminate, such that parts of the out-parameter have been assigned and others have not. However, no matter what their state, the values will be "safe" for memory-management purposes, that is, any out-parameters that were

successfully unmarshaled are autoreleased.

Exceptions and Return Values in Objective-C

For return values, the Objective-C mapping provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown.

See Also

- Operations
- Hello World Application
- Objective-C Mapping for Interfaces
- Objective-C Mapping for Exceptions
- Request Contexts

Objective-C Mapping for Classes

On this page:

Basic Objective-C Mapping for Classes

A Slice class is mapped similar to a structure and exception.

The generated class contains an instance variable and a property for each Slice data member. Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
}
```

The Slice compiler generates the following code for this definition:

Objective-C

```
@interface EXTimeOfDay : ICEObject
{
    ICEShort hour;
    ICEShort minute;
    ICEShort second;
}

@property(nonatomic, assign) ICEShort hour;
@property(nonatomic, assign) ICEShort minute;
@property(nonatomic, assign) ICEShort second;

-(id) init:(ICEShort)hour minute:(ICEShort)minute
second:(ICEShort)second;
+(id) timeOfDay;
+(id) timeOfDay:(ICEShort)hour minute:(ICEShort)minute
second:(ICEShort)second;
@end
```

There are a number of things to note about the generated code:

1. The generated class `EXTimeOfDay` derives from `ICEObject`, which is the parent of all classes. Note that `ICEObject` is *not* the same as `ICEObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The generated class contains a property for each Slice data member.
3. The generated class provides an `init` method that accepts one argument for each data member, and it provides the same two convenience constructors as structures and exceptions.

Derivation from `ICEObject` in Objective-C

All classes ultimately derive from a common base class, `ICEObject`. Note that this is not the same as implementing the `ICEObjectPrx` pr otocol (which is implemented by proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`ICEObject` defines a number of methods:

Objective-C

```
@protocol ICEObject <NSObject>
-(BOOL) ice_isA:(NSString*)typeId current:(ICECurrent*)current;
-(void) ice_ping:(ICECurrent*)current;
-(NSString*) ice_id:(ICECurrent*)current;
-(NSArray*) ice_ids:(ICECurrent*)current;
-(void) ice_dispatch:(id<ICERequest>)request;
@end

@interface ICEObject : NSObject <ICEObject, NSCopying>
-(BOOL) ice_isA:(NSString*)typeId;
-(void) ice_ping;
-(NSString*) ice_id;
-(NSArray*) ice_ids;
+(NSString*) ice_staticId;
-(void) ice_preMarshal;
-(void) ice_postUnmarshal;
-(id<ICESlicedData>) ice_getSlicedData;
@end
```

The methods are split between the `ICEObject` protocol and class because classes can be servants.

The methods of `ICEObject` behave as follows:

- `ice_isA`
  This method returns `YES` if the object supports the given type ID, and `NO` otherwise.
- `ice_ping`
  `ice_ping` provides a basic reachability test for the class. If it completes without raising an exception, the class exists and is reachable. Note that `ice_ping` is normally only invoked on the proxy for a class that might be remote because a class instance that is local (in the caller's address space) can always be reached.
- `ice_ids`
  This method returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This method returns the actual run-time type ID for a class. If you call `ice_id` via a pointer to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.

- `ice_staticId`
  This method returns the static type ID of a class.
- `ice_dispatch`
  This method dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.
- `ice_preMarshal`
  The Ice run time invokes this method prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this method after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.

Class Data Members in Objective-C

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding property.

Class Constructors in Objective-C

Classes provide the usual `init` method and a parameter-less convenience constructor that perform default initialization of the object's instance variables. These methods initialize the instance variables with zero-filled memory, with the following exceptions:

- A string data member is initialized to an empty string
- An enumerator data member is initialized to the first enumerator
- A structure data member is initialized to a default-constructed value

If you declare default values in your Slice definition, the `init` method and convenience constructor initialize each data member with its declared value instead.

In addition, if a class has data members, it provides an `init` method and a convenience constructor that accept one argument for each data member. This allows you to allocate and initialize a class instance in a single statement (instead of first having to allocate and default-initialize the instance and then assign to its properties).

For derived classes, the `init` method and the convenience constructor have one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order. For example:

Slice

```
class Base
{
    int i;
}

class Derived extends Base
{
    string s;
}
```

This generates:

Objective-C

```
@interface EXBase : ICEObject
// ...

@property(nonatomic, assign) ICEInt i;

-(id) init:(ICEInt)i;
+(id) base;
+(id) base:(ICEInt)i;
@end

@interface EXDerived : EXBase
// ...

@property(nonatomic, strong) NSString *s;

-(id) init:(ICEInt)i s:(NSString *)s;
+(id) derived;
+(id) derived:(ICEInt)i s:(NSString *)s;
@end
```

Derived Classes in Objective-C

Note that, in the preceding example, the derivation of the Slice definitions is preserved for the generated classes: `EXBase` derives from `ICEObject`, and `EXDerived` derives from `EXBase`. This allows you to treat and pass classes polymorphically: you can always pass an `EXDerived` instance where an `EXBase` instance is expected.

Passing Classes as Parameters in Objective-C

Classes are passed by pointer, like any other Objective-C object. For example, here is an operation that accepts a `Base` as an in-parameter and returns a `Derived`:

Slice

```
Derived getDerived(Base d);
```

The corresponding proxy method looks as follows:

Objective-C

```
-(EXDerived *) getDerived:(EXBase *)d;
```

To pass a null instance, you simply pass `nil`.

Operations of Classes in Objective-C

If you look back at the code that is generated for the `EXTimeOfDay` class, you will notice that there is no indication at all that the class has a `format` operation. As opposed to proxies, classes do not implement any protocol that would define which operations are available. This means that you can partially implement the operations of a class. For example, you might have a Slice class with five operations that is returned from a server to a client. If the client uses only one of the five operations, the client-side code needs to implement only that one operation and can leave the remaining four operations without implementation. (If the class were to implement a mandatory protocol, the client-side code would have to implement all operations in order to avoid a compiler warning.)

Of course, you must implement those operations that you actually intend to call. The mapping of operations for classes follows the *server-side* mapping for operations on interfaces: parameter types and labels are exactly the same. (See Parameter Passing in Objective-C for details.) In a nutshell, the server-side mapping is the same as the client-side mapping except that, for types that have mutable and immutable variants, they map to the immutable variant where the client-side mapping uses the mutable variant, and vice versa.

For example, here is how we could implement the `format` operation of our `TimeOfDay` class:

Objective-C

```
@interface TimeOfDayI : EXTimeOfDay
@end

@implementation TimeOfDayI
-(NSString *) format
{
    return [NSString stringWithFormat:@"%.2d:%.2d:%.2d",
self.hour, self.minute, self.second];
}
@end
```

By convention, the implementation of classes with operations has the same name as the Slice class with an `I`-suffix. Doing this is not mandatory — you can call your implementation class anything you like. However, if you do not want to use the `I`-suffix naming, we recommend that you adopt another naming convention and follow it consistently.

Note that `TimeOfDayI` derives from `EXTimeOfDay`. This is because, as we will see in a moment, the Ice run time will instantiate a `TimeOfDayI` instance whenever it receives a `TimeOfDay` instance over the wire and expects that instance to provide the properties of `EXTimeOfDay`.

Class Factories in Objective-C

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

Slice

```
interface Time
{
    TimeOfDay get();
}
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDayI` class. However, unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements a `format` method. To allow the Ice run time to instantiate the correct object, we must provide a factory that knows that the Slice `TimeOfDay` class is implemented by our `TimeOfDayI` class.

To supply the Ice run time with a value factory for our `TimeOfDayI` class, we must implement the `ValueFactory` interface:

Slice

```
module Ice
{
 ["delegate"]
    local interface ValueFactory
    {
        Value create(string type);
    }
}
```

In Objective-C this interface is generated as `block`. The Ice run time calls the value factory block when it needs to instantiate a `TimeOfDay` class. Here's a possible implementation of our value factory:

Objective-C

```
ICEValueFactory factory = ^ICEObject* (NSString* type)
NS_RETURNS_RETAINED
{
    NSAssert([type isEqualToString:@"::Example::TimeOfDay"]);
    return [[TimeOfDayI alloc] init];
}
@end
```

The `block` is passed the type ID of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::Example::TimeOfDay"`. Our implementation checks the type ID: if it matches `"::Example::TimeOfDay"`, it instantiates and returns a `TimeOfDayI` object. For other type IDs, it asserts because it does not know how to instantiate other types of objects.

When using automatic reference counting (ARC), you should specify `NS_RETURNS_RETAINED` in the block definition. It's not needed if you're not using ARC but your factory *must not* autorelease the returned instance. The Ice run time takes care of the necessary memory management activities on your behalf.

Given a factory implementation, such as the one we assigned to `factory`, we must inform the Ice run time of the existence of the factory:

Objective-C

```
id<ICECommunicator> ice = ...;
[[ic getValueFactoryManager]
add:factory sliceId:@"::Example::TimeOfDay"];
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::Example::TimeOfDay"`, it invokes our block, which returns a `TimeOfDayI` instance to the Ice run time.

Finally, keep in mind that if a class has only data members, but no operations, you do not need to create and register a value factory to receive instances of such a class. You're only required to register a value factory when a class has operations.

Using a Category to Implement Operations in Objective-C

An alternative to registering a value factory is to use an Objective-C category to implement operations. For example, we could have implemented our `format` method using a category instead:

Objective-C

```
@interface EXTimeOfDay (TimeOfDayI)
@end

@implementation EXTimeOfDay (TimeOfDayI)
-(NSString *) format
{
    return [NSString stringWithFormat:@"%.2d:%.2d:%.2d",
self.hour, self.minute, self.second];
}
@end
```

In this case, there is no need to derive from the generated `EXTimeOfDay` class because we provide the `format` implementation as a category. There is also no need to register a value factory: the Ice run time instantiates an `EXTimeOfDay` instance when a `TimeOfDay` instance arrives over the wire, and the `format` method is found at run time when it is actually called.

This is a viable alternative approach to implementing class operations. However, keep in mind that, if the operation implementation requires use of instance variables that are not defined as part of the Slice definitions of a class, you cannot use this approach because Objective-C categories do not permit you to add instance variables to a class.

Copying of Classes in Objective-C

Classes implement `NSCopying`. The behavior is the same as for structures, that is, the copy is shallow. To illustrate this, consider the following class definition:

Slice

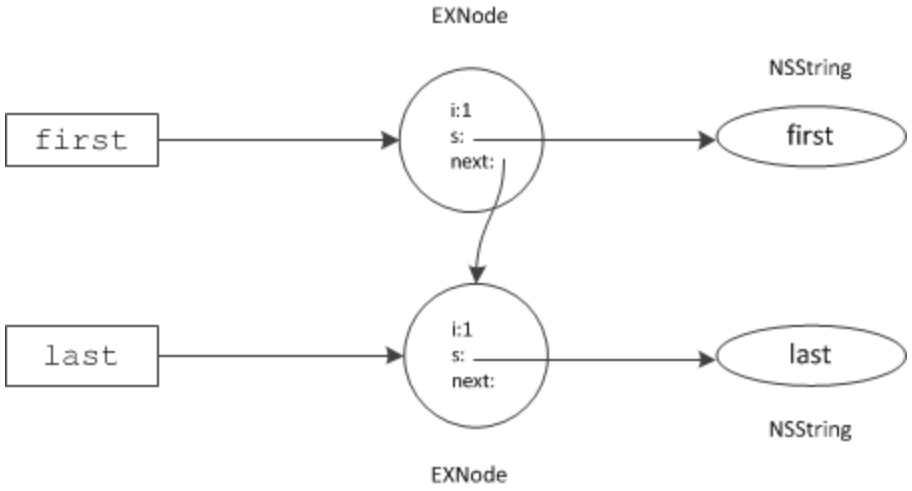
```
class Node
{
    int i;
    string s;
    Node next;
}
```

We can initialize two instances of type `EXNode` as follows:

Objective-C

```
NSString lastString = [NSString stringWithString:@"last"];
EXNode *last = [EXNode node:99 s:lastString next:nil];

NSString firstString = [NSString stringWithString:@"first"];
EXNode *first = [EXNode node:1 s:firstString next:last];
```

This creates the situation shown below:



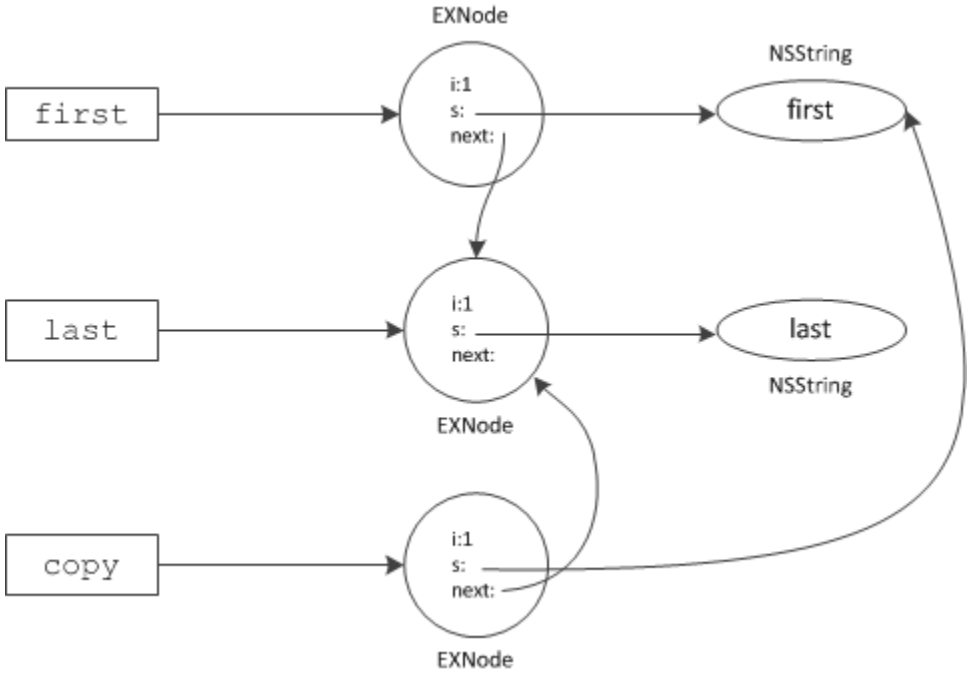
Two instances of type `EXNode`.

Now we create a copy of the first node by calling `copy`:

Objective-C

```
EXNode *copy = [first copy];
```

This creates the situation shown here:



`EXNode` instances after calling `copy` on `first`.

As you can see, the first node is copied, but the last node (pointed at by the `next` instance variable of the first node) is not copied; instead, `first` and `copy` now both have their `next` instance variable point at the same last node, and both point at the same string.

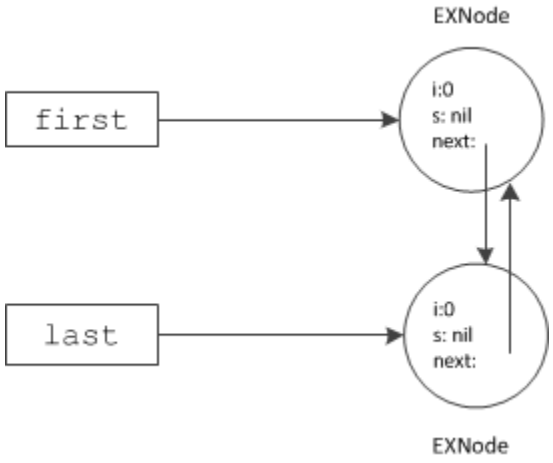
Cyclic References in Objective-C

One thing to be aware of are cyclic references among classes. As an example, we can easily create a cycle by executing the following statements:

Objective-C

```
EXNode *first = [EXNode node];
ExNode *last = [EXNode node];
first.next = last;
last.next = first;
```

This makes the `next` instance variable of the two classes point at each other, creating the cycle shown below:



Two nodes with cyclic references.

There is no problem with sending this class graph as a parameter. For example, you could pass either `first` or `last` as a parameter to an operation and, in the server, the Ice run time will faithfully rebuild the corresponding graph, preserving the cycle. However, if a server returns such a graph from an operation invocation as the return value or as an out-parameter, all class instances that are part of a cycle are leaked. The same is true on the client side: if you receive such a graph from an operation invocation and do not explicitly break the cycle, you will leak all instances that form part of the cycle.

Because it is difficult to break cycles manually (and, on the server side, for return values and out-parameters, it is impossible to break them), we recommend that you avoid cyclic references among classes.

See Also

- Simple Classes
- Server-Side Objective-C Mapping for Interfaces
- Parameter Passing in Objective-C
- Dispatch Interceptors
- Value Factories

Objective-C Mapping for Interfaces by Value

Slice permits you to pass an interface by value:

Slice

```
interface ClassBase
{
    void someOp();
    // ...
}

interface Processor
{
    ClassBase process(ClassBase b);
}

class SomeClass implements ClassBase
{
    // ...
}
```

Note that `process` accepts and returns a value of type `ClassBase`. This is *not* the same as passing `id<ClassBasePrx>`, which is a *proxy* to an object of type `ClassBase` that is possibly remote. Instead, what is passed here is an *interface*, and the interface is passed by *value*.

The immediate question is "what does this mean?" After all, interfaces are abstract and, therefore, it is impossible to pass an interface by value. The answer is that, while an interface cannot be passed, what *can* be passed is a class that implements the interface. That class is type-compatible with the formal parameter type and, therefore, can be passed by value. In the preceding example, `SomeClass` implements `ClassBase` and, hence, can be passed to and returned from the `process` operation.

The Objective-C mapping maps interface-by-value parameters to `ICEObject*`, regardless of the type of the interface. For example, the proxy protocol for the `process` operation is:

Objective-C

```
-(ICEObject *) process:(ICEObject *)b;
```

This means that you can pass a class of any type to the operation, even if it is not type-compatible with the formal parameter type, because all classes derive from `ICEObject`. However, an invocation of `process` is still type-safe at run time: the Ice run time verifies that the class instance that is passed implements the specified interface; if not, the invocation throws an `ICEMarshalException`.

Passing interfaces by value as `ICEObject*` is a consequence of the decision to not generate a formal protocol for classes. (If such a protocol would exist, the formal parameter type could be `id<`*`ProtocolName`*`>`. However, as we described for the class mapping, a protocol would require the implementation of a class to implement all of its operations, which can be inconvenient. Because it is rare to pass interfaces by value (more often, the formal parameter type will be a base *class* instead of a base *interface*), the minor loss of static type safety is an acceptable trade-off.

See Also

- Passing Interfaces by Value
- Objective-C Mapping for Classes

Objective-C Mapping for Optional Data Members

The Objective-C mapping for optional data members in Slice classes and exceptions adds an extra boolean instance variable as well as two selectors for testing if the optional is set and clearing its value. The argument for the optional data member in the convenience constructor is mapped according to the optional parameter mapping for Slice operations. Consider the following Slice definition:

Slice

```
class C
{
    string name;
    optional(2) string alternateName;
    optional(5) bool active;
}
```

The generated Objective-C code provides the following API:

Objective-C

```
@interface testC : ICEObject
{
    NSString *name;
    NSString *alternateName;
    BOOL has_alternateName__;
    BOOL active;
    BOOL has_active__;
}
@property(nonatomic, strong) NSString *name;
@property(nonatomic, strong) NSString *alternateName;
@property(nonatomic, assign) BOOL active;
-(id) init:(NSString*)name alternateName:(id)alternateName
active:(id)active;
+(id) c:(NSString*)name alternateName:(id)alternateName
active:(id)active;
+(id) c;
-(void)setAlternateName:(NSString*)alternateName;
-(BOOL)hasAlternateName;
-(void)clearAlternateName;
-(void)setActive:(BOOL)active;
-(BOOL)hasActive;
-(void)clearActive;
@end
```

Note that calling a `get` method or accessing the property when the value is not currently set returns an undefined value.

See Also

- Optional Data Members

Asynchronous Method Invocation (AMI) in Objective-C

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

On this page:

Basic Asynchronous API in Objective-C

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Proxy Methods for AMI in Objective-C

Besides the synchronous proxy methods, the Objective-C mapping generates the following asynchronous proxy methods:

Objective-C

```
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
context:(ICEContext *)context;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
response:(void(^)(NSMutableString*))response
exception:(void(^)(ICEException*))exception;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
context:(ICEContext *)context
response:(void(^)(NSMutableString*))response
exception:(void(^)(ICEException*))exception;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
response:(void(^)(NSMutableString*))response
exception:(void(^)(ICEException*))exception
sent:(void(^)(BOOL))sent;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
context:(ICEContext *)context
response:(void(^)(NSMutableString*))response
exception:(void(^)(ICEException*))exception
sent:(void(^)(BOOL))sent;

-(NSMutableString *) end_getName:(id<ICEAsyncResult>)result;
```

As you can see, the single `getName` operation results in several `begin_getName` methods as well as an `end_getName` method. The `begin_` methods optionally accept a per-invocation context and callbacks.

- The `begin_getName` methods send (or queue) an invocation of `getName`. These methods do not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

Objective-C

```
id<EXEmployeesPrx> e = [EXEmployeesPrx checkedCast:...];
id<ICEAsyncResult> r = [e begin_getName:99]

// Continue to do other things here...

NSString* name = [e end_getName:r];
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `id<ICEAsyncResult>`. This value contains the state that the Ice run time requires to

keep track of the asynchronous invocation. You must pass the `id<ICEAsyncResult>` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. The `end_` method accepts the `id<ICEAsyncResult>` object as its only argument and returns the out-parameters using the same semantics as for regular synchronous invocations. For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

Objective-C

```
-(id<ICEAsyncResult>) begin_op:(ICEInt)inp1 inp2:(NSString *)inp2;
-(ICEDouble) end_op:(BOOL*)outp1 outp2:(ICELong*)outp2
result:(id<ICEAsyncResult>)result;
```

The call to `end_op` returns the out-parameters as follows:

Objective-C

```
BOOL outp1;
ICELong outp2;
ICEDouble doubleValue = [p end_op:&outp1 outp2:&outp2 result:result];
```

Exception Handling for AMI in Objective-C

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `ICECommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `NSException` with the `NSInvalidArgumentException` name. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_ method`, or if the `id<ICEAsyncResult>` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

The `ICEAsyncResult` Protocol in Objective-C

The `id<ICEAsyncResult>` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

Objective-C

```
@protocol ICEAsyncResult <NSObject>
-(id<ICECommunicator>) getCommunicator;
-(id<ICEConnection>) getConnection;
-(id<ICEObjectPrx>) getProxy;

-(BOOL) isCompleted;
-(void) waitForCompleted;

-(BOOL) isSent;
-(void) waitForSent;

-(BOOL) sentSynchronously;
-(NSString*) getOperation;
@end
```

The methods have the following semantics:

- `getCommunicator`
  This method returns the communicator that sent the invocation.
- `getConnection`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `getConnection` method only returns a non-nil value when the `ICEAsyncResult` is obtained by calling `begin_flushBatchRequests` on a `Connection` object.
- `getProxy`
  This method returns the proxy that was used to call the `begin_` method, or nil if the `ICEAsyncResult` was not obtained via an asynchronous proxy invocation.
- `getOperation`
  This method returns the name of the operation.
- `isCompleted`
  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.
- `waitForCompleted`
  This method blocks the caller until the result of an invocation becomes available.
- `isSent`
  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued, `isSent` returns false.
- `waitForSent`
  This method blocks the calling thread until a request has been written to the client-side transport.
- `sentSynchronously`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

Polling for Completion in Objective-C

The `ICEAsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the

following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

Objective-C

```
NSInputStream* stream = ...
id<EXFileTransferPrx> ft = [EXFileTransferPrx checkedCast:...];
int chunkSize = ...;
int offset = 0;
while([stream hasBytesAvailable])
{
    char bytes[chunkSize];
    int l = [stream read:bytes maxLength:sizeof(bytes)];
    if(l > 0)
    {
        [ft send:offset bytes:[ByteSeq dataWithBytes:bytes length:l]];
        offset += l;
    }
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

Objective-C

```
NSInputStream* stream = ...
id<EXFileTransferPrx> ft = [EXFileTransferPrx checkedCast:...];
int chunkSize = ...;
int offset = 0;
NSMutableArray* results = [NSMutableArray arrayWithCapacity:5];
int numRequests = 5;
while([stream hasBytesAvailable])
{
    char bytes[chunkSize];
    int l = [stream read:bytes maxLength:sizeof(bytes)];
    if(l > 0)
    {
        // Send up to numRequests + 1 chunks asynchronously.
        id<ICEAsyncResult> r =
            [ft begin_send:offset bytes:[ByteSeq dataWithBytes:bytes
length:l]];
        offset += l;

        // Wait until this request has been passed to the
        // transport.
        [r waitForSent];
        [results addObject:r];

        // Once there are more than numRequests, wait for the
        // least recent one to complete.
        while([results count] > numRequests)
        {
            r = [results objectAtIndex:0];
            [results removeObjectAtIndex:0];
            [r waitForCompleted];
        }
    }
}

// Wait for any remaining requests to complete.
for(id<ICEAsyncResult> r in results)
{
    [r waitForCompleted];
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Completion Callbacks in Objective-C

The `begin_` method accepts three optional callback arguments that allow you to be notified asynchronously when a request completes. Here is the signature of the `begin_getName` method that we saw earlier:

Objective-C

```
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
    response:(void(^)(NSMutableString*))response
    exception:(void(^)(ICEException*))exception;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
    context:(ICEContext *)context
    response:(void(^)(NSMutableString*))response
    exception:(void(^)(ICEException*))exception;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
    response:(void(^)(NSMutableString*))response
    exception:(void(^)(ICEException*))exception
    sent:(void(^)(BOOL))sent;

-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
    context:(ICEContext *)context
    response:(void(^)(NSMutableString*))response
    exception:(void(^)(ICEException*))exception
    sent:(void(^)(BOOL))sent;
```

The value you pass for the response callback (`response`), the exception callback (`exception`), or the sent callback (`sent`) argument must be an Objective-C block. The response callback is invoked when the request completes successfully, and the exception callback is invoked when the operation raises an exception. (The sent callback is primarily used for flow control.)

For example, consider the following callbacks for an invocation of the `getName` operation:

Objective-C

```
void(^getNameCB)(NSMutableString*) = ^(NSMutableString* name)
{
    NSLog(@"Name is: %@", name);
};

void(^failureCB)(ICEException*) = ^(ICEException* ex)
{
    NSLog(@"Exception is: %@", [ex description]);
};
```

The response callback parameters depend on the operation signature. If the operation has a non-`void` return type, the first parameter of the response callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

The exception callback is called if the invocation fails because of an Ice run time exception, or if the operation raises a user exception.

To inform the Ice run time that you want to receive callbacks for the completion of the asynchronous call, you pass the callbacks to the `begi`

n_ method:

Objective-C

```
e = [EmployeesPrx checkedCast:...]

[e begin_getName:99 response:getNameCB exception:failureCB];
```

You can also pass the Objective-C blocks directly to the call:

Objective-C

```
[e begin_getName:99
    response: ^(NSMutableString* name)
        {
            NSLog(@"Name is: %@", name);
        }
    exception: ^(ICEException* ex)
        {
            NSLog(@"Exception is: %@", [ex description]);
        }];
```

Ice enforces the following semantics at run time regarding which callbacks can be optionally specified with a `nil` value:

- You must supply an exception callback.
- You may omit the response callback for an operation that returns no data (that is, an operation with a void return type and no out-parameters).

Memory Management
The Ice run time creates an `NSAutoReleasePool` object before dispatching a completion callback. The pool is released once the dispatch is complete

Oneway Invocations in Objective-C

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws `NSException` with the `NSInvalidArgumentException` name.

The callback signatures look exactly as for a twoway invocation, but the response block is never called and may be `nil`.

Flow Control in Objective-C

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `[ICEAsyncResult sentSynchronously]` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `[ICEAsyncResult sentSynchronously]` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

You can supply a sent callback to be notified when the request was successfully sent:

Objective-C

```
void(^sentCB)(BOOL) = ^(BOOL sentSynchronously)
{
    ...
}
```

You inform the Ice run time that you want to be notified when a request has been passed to the local transport as usual:

Objective-C

```
[e begin_getName:99 response:getNameCB exception:failureCB sent:sentCB];
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the sent callback from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the sent callback from a different thread once it has written the request to the local transport. The boolean `sentSynchronously` parameter indicates whether the request was sent synchronously or was queued.

The sent callback allows you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Batch Requests in Objective-C

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws `NSException` with the `NSInvalidArgumentException` name.

A batch oneway invocation never calls the callbacks unless an error occurs before the request is queued. The returned `ICEAsyncResult` fo r a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The returned result can also be marked completed if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `[ICEAsyncResult getConnection]`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency in Objective-C

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the sent callback from the thread calling the `begin_` method if the request could be sent synchronously. In the sent callback, you know which thread is calling the callback by looking at the `sentSynchronously` parameter.

See Also

- Request Contexts
- Batched Invocations

slice2objc Command-Line Options

The Slice-to-Objective-C compiler, `slice2objc`, offers the following command-line options in addition to the standard options:

- `--include-dir DIR`
  Modifies `#import` directives in source files to prepend the path name of each header file with the directory *`DIR`*.

See Also

- Using the Slice Compilers

Example of a File System Client in Objective-C

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The Objective-C code shown here hardly differs from the code you would write for an ordinary Objective-C program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Objective-C object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side Objective-C mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
["objc:prefix:FS"]
module Filesystem
{
    exception GenericError
    {
        string reason;
    }

    interface Node
    {
        idempotent string name();
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

Objective-C

```
#import <objc/Ice.h>
#import <Filesystem.h>
#import <stdio.h>

static void
listRecursive(id<FSDirectoryPrx> dir, int depth)
{
    // ...
}

int
main(int argc, char* argv[])
{
    int status = EXIT_FAILURE;
    @autoreleasepool
    {
        id<ICECommunicator> communicator = nil;
        @try
        {
            communicator = [ICEUtil createCommunicator:&argc argv:argv];
            //
            // Create a proxy for the root directory
            //
            id<FSDirectoryPrx> rootDir = [FSDirectoryPrx checkedCast:
            [communicator stringToProxy:@"RootDir:default -p 10000"]];
            if(!rootDir)
            {
                [NSException raise:@"invalid proxy" format:@"nil"];
            }

            //
            // Recursively list the contents of the root directory
            //
            printf("Contents of root directory:\n");
            listRecursive(rootDir, 0);

            status = EXIT_SUCCESS;
        }
        @catch (NSException *ex)
        {
            NSLog(@"%@\n", ex);
        }

        [communicator destroy];
    }
    return status;
}
```

1. The code imports a few header files:
    - `obj/Ice.h`: Always included in both client and server source files, provides definitions that are necessary for accessing

the Ice run time.
- `Filesystem.h` The header that is generated by the Slice compiler from the Slice definitions in `Filesystem.ice`.
- `stdio.h`: The implementation of `listRecursive` prints to `stdout`.

2. The structure of the code in `main` follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
3. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`:

Objective-C

```
// Print the specified number of tabs.
static void
printIndent(int depth)
{
    while(depth-- > 0)
    {
        putchar('\t');
    }
}

// Recursively print the contents of directory "dir" in tree fashion.
// For files, show the contents of each file. The "depth"
// parameter is the current nesting level (for indentation).

static void
listRecursive(id<FSDirectoryPrx> directory, int depth)
{
    ++depth;
    FSNodeSeq *contents = [directory list];

    for(id<FSNodePrx> node in contents)
    {
        id<FSDirectoryPrx> dir = [FSDirectoryPrx checkedCast:node];
        id<FSFilePrx> file = [FSFilePrx uncheckedCast:node];
        printIndent(depth);
        printf("%s%s\n", [[node name] UTF8String], (dir ? "
(directory):" : " (file):"));
        if(dir)
        {
            listRecursive(dir, depth);
        }
        else
        {
            FSLines *text = [file read];
            for(NSString *line in text)
            {
                printIndent(depth);
                printf("\t%s\n", [line UTF8String]);
            }
        }
    }
}
```

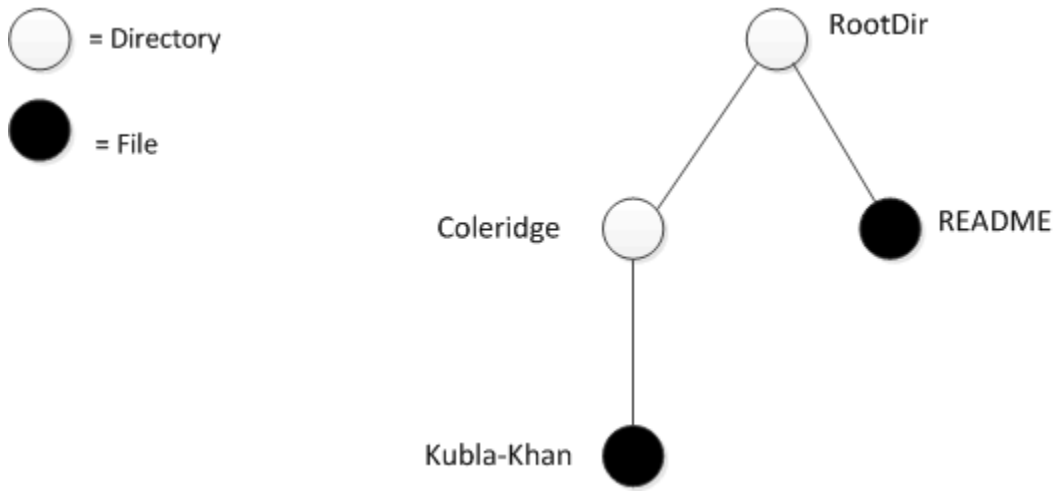
The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `No`

de proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `id<FSDirectoryPrx>` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the `Node` *is-a* `File` and, therefore, an `uncheckedCast` is sufficient to get an `id<FSFilePrx>`.
In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
    - If it is a directory, the code recurses, incrementing the indent level.
    - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Example of a File System Server in Objective-C
- Object Life Cycle
- IceGrid

Server-Side Slice-to-Objective-C Mapping

The mapping for Slice data types to Objective-C is identical on the client side and server side, except for operation parameters, which map slightly differently for types that have mutable and immutable variants (strings, sequence, and dictionaries). This means that the mappings in the Client-Side Slice-to-Objective-C Mapping also apply to the server side. However, for the server side, there are a few additional things you need to know — specifically, how to:

- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time.

Although the examples we present are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Objective-C mapping rules beyond those described here.

Topics

Server-Side Objective-C Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing methods in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in Objective-C

On the client side, interfaces map to proxy protocols and classes. On the server side, interfaces map to *skeleton* protocols and classes. A skeleton is a class that has a method for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
["objc:prefix:FS"]
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
// ...
}
```

The Slice compiler generates the following definition for this interface:

Objective-C

```
@protocol FSNode <ICEObject>
-(NSString *) name:(ICECurrent *)current;
@end

@interface FSNode : ICEObject
// ...
@end
```

As you can see, the server-side API consists of a protocol and a class, known as the *skeleton protocol* and *skeleton class*. The methods of the skeleton class are internal to the mapping, so they do not concern us here. The skeleton protocol defines one method for each Slice operation. As for the client-side mapping, the method name is the same as the name of the corresponding Slice operation. If the Slice operation has parameters or a return value, these are reflected in the generated method, just as they are for the client-side mapping. In addition, each method has an additional trailing parameter of type `ICECurrent`. This parameter provides additional information about an invocation to your server-side code.

As for the client-side mapping, the generated code reflects the fact that all Slice interfaces and classes ultimately derive from `Ice::Object`. As you can see, the generated protocol incorporates the `ICEObject` protocol, and the generated class derives from the `ICEObject` class.

Servant Classes in Objective-C

The Objective-C mapping supports two different ways to implement servants. You can implement a servant by deriving from the skeleton class and implementing the methods for the Slice operations in your derived class. Alternatively, you can use a delegate servant, which need not derive from the skeleton class.

Derived Servants in Objective-C

To provide an implementation for an Ice object, you can create a servant class that derives from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

Objective-C

```
@interface NodeI : FSNode <FSNode>
{
    @private
        NSString *myName;
}

+(id) nodei:(NSString *)name;
@end
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.)

Note that `NodeI` derives from `FSNode`, that is, it derives from its skeleton class. In addition, it adopts the `FSNode` protocol. Adopting the protocol is not strictly necessary; however, if you do write your servants this way, the compiler emits a warning if you forget to implement one or more Slice operations for the corresponding interface, so we suggest that you make it a habit to always have your servant class adopt its skeleton protocol.

As far as Ice is concerned, the `NodeI` class must implement the single `name` method that is defined by its skeleton protocol. That way, the Ice run time gets a servant that can respond to the operation that is defined by its Slice interface. You can add other methods and instance variables as you see fit to support your implementation. For example, in the preceding definition, we added a `myName` instance variable and property and a convenience constructor. Not surprisingly, the convenience constructor initializes the `myName` instance variable and the `name` method returns the value of that variable:

Objective-C

```
@implementation NodeI

+(id) nodei:(NSString *)name
{
    NodeI *instance = [[NodeI alloc] init];
    instance.myName = [name copy];
    return instance;
}

-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

@end
```

Delegate Servants in Objective-C

An alternate way to implement a servant is to use a delegate. `ICEServant` provides two constructors to do this:

Objective-C

```
@interface ICEServant : ICEObject
-(id) initWithDelegate:(id)delegate;
+(id) objectWithDelegate:(id)delegate;
...
@end
```

The `delegate` parameter specifies an object to which the servant will delegate operation invocations. That object need not derive from the skeleton class; the only requirement on the delegate is that it must have an implementation of the methods corresponding to the Slice operations that are called by clients. As for derived servants, we suggest that the delegate adopt the skeleton protocol, so the compiler will emit a warning if you forget to implement one or more Slice operations in the delegate.

The implementation of the Slice operations in a delegate servant is exactly the same as for a derived servant.

Delegate servants are useful if you need to derive your servant implementation from a base class in order to access some functionality. In that case, you cannot also derive the servant from the generated skeleton class. A delegate servant gets around Objective-C's single inheritance limitation and saves you having to write a servant class that forwards each operation invocation to the delegate.

Another use case are different interfaces that share their implementation. As an example, consider the following Slice definitions:

Slice

```
interface Intf1
{
    void op1();
}

interface Intf2
{
    void op2();
}
```

If `op1` and `op2` are substantially similar in their implementation and share common state, it can be convenient to implement the servants for `Intf1` and `Intf2` using a common delegate class:

Objective-C

```
@interface Intf1AndIntf2 : NSObject<EXIntf1, EXIntf2>
    +(id) intf1AndIntf2;
@end

@implementation Intf1AndIntf2
    +(id) intf1AndIntf2 { /*...*/ }
    -(void) op1:(ICECurrent*)current { /*...*/ }
    -(void) op2:(ICECurrent*)current { /*...*/ }
@end
```

See Instantiating an Objective-C Servant for an example of how to instantiate delegate servants.

Delegate servants do not permit you to override operations that are inherited from `ICEObject` (such as `ice_ping`). Therefore, if you want to override `ice_ping`, for example, to implement a default servant, you must use a derived servant.

Memory Management

The Ice run time creates an `NSAutoReleasePool` object before dispatching server-side invocations. The pool is released once the dispatch is complete.

See Also

- Slice for a Simple File System
- Objective-C Mapping for Interfaces
- Parameter Passing in Objective-C
- Object Incarnation in Objective-C
- The Current Object
- Default Servants

Parameter Passing in Objective-C

This page shows how to implement parameters for Slice operations in Objective-C.

On this page:

Implementing Parameters for Slice Operations in Objective-C

For each parameter of a Slice operation, the Objective-C mapping generates a corresponding parameter for the method in the skeleton. In addition, every method has an additional, trailing parameter of type `ICECurrent`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method of the `Node` skeleton protocol has a single parameter of type `ICECurrent`. We will ignore this parameter for now.

Parameter passing on the server side follows the rules for the client side (with one exception):

- In-parameters and the return value are passed by value or by pointer, depending on the parameter type.
- Out-parameters are passed by pointer-to-pointer.

The exception to the client-side rules concerns types that come in mutable and immutable variants (strings, sequences, and dictionaries). For these, the server-side mapping passes the mutable variant where the client-side passes the immutable variant, and vice versa.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
interface Intf
{
    string op(string sin, out string sout);
}
```

The generated skeleton protocol for this interface looks as follows:

Objective-C

```
@protocol EXIntf <ICEObject>
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current;
@end
```

As you can see, the in-parameter `sin` is of type `NSMutableString`, and the out parameter and return value are passed as `NSString` (the opposite of the client-side mapping). This means that in-parameters are passed to the servant as their mutable variant, and it is safe for you to modify such in-parameters. This is useful, for example, if a client passes a sequence to the operation, and the operation returns the sequence with a few minor changes. In that case, there is no need for the operation implementation to copy the sequence. Instead, you can simply modify the passed sequence as necessary and return the modified sequence to the client.

Here is an example implementation of the operation:

Objective-C

```
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current
{
    printf("%s\n", [sin UTF8String]); // In-params are initialized
    *sout = [sin appendString:@"appended"]; // Assign out-param
    return @"Done";                         // Return a string
}
```

Memory Management for Operations in Objective-C

If you are not using ARC, to avoid leaking memory, you must be aware of how the Ice run time manages memory for operation implementations:

- In-parameters passed to the servant are already autoreleased.
- Out-parameters and return values must be returned by the servant as autoreleased values.

This follows the usual Objective-C convention: the allocator of a value is responsible for releasing it. This is what the Ice run time does for in-parameters, and what you are expected to do for out-parameters and return values. These rules also mean that it is safe to return an in-parameter as an out-parameter or return value. For example:

Objective-C

```
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current
{
    *sout = sin; // Works fine.
    return sin;  // Works fine.
}
```

The Ice run time creates and releases a separate autorelease pool for each invocation. This means that the memory for parameters is reclaimed as soon as the run time has marshaled the operation results back to the client.

Thread-Safe Marshaling in Objective-C

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For Objective-C applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following Slice:

Slice

```
sequence<int> IntSeq;
sequence<IntSeq> IntIntSeq;
sequence<string> StringSeq;
class Grid
{
    StringSeq xLabels;
    StringSeq yLabels;
    IntIntSeq values;
}

interface GridIntf
{
   Grid getGrid();
   void clearValues();
}
```

And the following servant implementation:

Objective-C

```
-(Grid*) getGrid:(ICECurrent *)current
{
    Grid* r;
    @synchronized(self)
    {
        r = grid;
    }
    return r;
}

-(void) clearValues:(ICECurrent *)current
{
    @synchronized(self)
    {
  if([grid.values isKindOfClass:[NSMutableArray class]])
        {
            [(NSMutableArray*)grid.values removeAllObjects];
        }
        else
        {
            grid.values = [MutableIntIntSeq array];
        }
    }
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned class in preparation to send a reply

message, it is possible for another thread to dispatch the `clearValues` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `clearValues` operations does not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `clearValues` replaces `grid` with a copy that contains empty values, leaving the previous contents of `grid` unchanged:

Objective-C

```
-(void) clearValues:(ICECurrent *)current
{
    @synchronized(self)
    {
        grid.values = [MutableIntIntSeq array];
    }
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because its members are never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

See Also

- Server-Side Objective-C Mapping for Interfaces
- Raising Exceptions in Objective-C
- The Current Object

Raising Exceptions in Objective-C

To throw an exception from an operation implementation, you simply allocate the exception, initialize it, and throw it. For example:

Objective-C

```
-(void) write:(NSMutableArray *)text current:(ICECurrent *)current
{
    // Try to write the file contents here...
    // Assume we are out of space...
    if(error)
    {
        @throw [FSGenericError genericError:@"file too large"];
    }
}
```

As for out-parameters and return values, you must take care to throw an autoreleased exception if you are not using ARC.

If you throw an arbitrary Objective-C exception that does not derive from `ICEException`, the client receives an `UnknownException`.

The server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw an Ice run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

See Also

- Run-Time Exceptions
- Objective-C Mapping for Exceptions
- Server-Side Objective-C Mapping for Interfaces
- Parameter Passing in Objective-C

Object Incarnation in Objective-C

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must follow these steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating an Objective-C Servant

Instantiating a servant means to allocate an instance on the heap:

Objective-C

```
NodeI *servant = [NodeI nodei:@"Fred"];
```

This code creates a new `NodeI` instance. For this example, we used the convenience constructor we saw earlier. Of course, you are not obliged to define such a constructor but, if you do not, you must explicitly call `release` or `autorelease` on the servant.

For a delegate servant, instantiation would look as follows:

Objective-C

```
Intf1AndIntf2 *delegate = [Intf1AndIntf2 intf1AndIntf2];
ICEObject *servant = [ICEServant objectWithDelegate:delegate];
```

Creating an Identity in Objective-C

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
["objc:prefix:ICE"]
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

Objective-C

```
ICEIdentity ident = [ICEIdentity identity:"Fred" category:nil];
```

Activating an Objective-C Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `adapter` variable, we can write:

Objective-C

```
[adapter add:servant identity:ident];
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant pointer from the servant map and dispatches the incoming request into the correct method on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

Putting the preceding points together, we can write a simple method that instantiates and activates one of our `NodeI` servants. For this example, we use a simple method on our servant called `activate` that activates a servant in an object adapter with the passed identity:

Objective-C

```
-(void) activate:(id<ICEObjectAdapter>)a name:(NSString *)name
{
    ICEIdentity ident = [ICEIdentity identity:name category:nil];
    [a add:self identity:ident];
}
```

UUIDs as Identities in Objective-C

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. The `ICEUtil` class contains a helper function to create such identities:

Objective-C

```
@interface ICEUtil : NSObject
+(id) generateUUID;
// ...
@end
```

When executed, this method returns a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones. You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step:

Objective-C

```
-(id<ICEObjectPrx>) addWithUUID:(ICEObject*)servant
```

Note that the operation returns the proxy for the servant just activated.

Creating Proxies in Objective-C

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in Objective-C

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object, as we saw earlier. This means we can write:

Objective-C

```
NodeI *servant = [NodeI nodei:name];
id<FSNodePrx> proxy = [FSNodePrx uncheckedCast:[adapter addWithUUID:ser
vant]];

// Pass proxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `id<ICEObjectPrx>`.

Direct Proxy Creation in Objective-C

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
["objc:prefix:ICE"]
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

Objective-C

```
ICEIdentity *ident = [ICEIdentity identity];
ident.name = [ICEUtil generateUUID];
id<ICEObjectPrx> o = [adapter createProxy:ident];
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Writing an Ice Application with Objective-C
- Server-Side Objective-C Mapping for Interfaces
- Object Adapter States
- Servant Locators
- Object Life Cycle

Example of a File System Server in Objective-C

This page presents the source code for a C++ server that implements our file system and communicates with the client we wrote earlier. The code here is fully functional, apart from the required interlocking for threads.

The server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

On this page:

Implementing a File System Server in Objective-C

We have now seen enough of the server-side Objective-C mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is composed of three source files:

- `Server.m`
  This file contains the server main program.
- `FileI.m`
  This file contains the implementation for the `File` servants.
- `DirectoryI.m`
  This file contains the implementation for the `Directory` servants.

Server main Program in Objective-C

Our server main program, in the file `Server.m`, uses the structure we saw earlier:

Objective-C

```
#import <objc/Ice.h>
#import <FileI.h>
#import <DirectoryI.h>

int
main(int argc, char* argv[])
{
    int status = EXIT_FAILURE;
    @autoreleasepool
    {
        id<ICECommunicator> communicator = nil;
        @try
        {
            communicator = [ICEUtil createCommunicator:&argc argv:argv];

            id<ICEObjectAdapter> adapter = [communicator
createObjectAdapterWithEndpoints:@"SimpleFilesystem"
```

```
endpoints:@"default -p 10000"];

            //
            // Create the root directory (with name "/" and no parent)
            //
            DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
            [root activate:adapter];

            //
            // Create a file called "README" in the root directory
            //
            FileI *file = [FileI filei:@"README" parent:root];
            NSMutableArray *text = [NSMutableArray
arrayWithObject:@"This file system contains a collection of poetry."];
            [file write:text current:nil];
            [file activate:adapter];

            //
            // Create a directory called "Coleridge" in the root
directory
            //
            DirectoryI *coleridge = [DirectoryI directoryi:@"Coleridge"
parent:root];
            [coleridge activate:adapter];

            //
            // Create a file called "Kubla_Khan" in the Coleridge
directory
            //
            file = [FileI filei:@"Kubla_Khan" parent:coleridge];
            text = [NSMutableArray arrayWithObjects:@"In Xanadu did
Kubla Khan",
                                                    @"A stately
pleasure-dome decree:",
                                                    @"Where Alph, the
sacred river, ran",
                                                    @"Through caverns
measureless to man",
                                                    @"Down to a sunless
sea.", nil];
            [file write:text current:nil];
            [file activate:adapter];

            //
            // All objects are created, allow client requests now
            //
            [adapter activate];

            //
            // Wait until we are done
```

```
            //
            [communicator waitForShutdown];
            status = EXIT_SUCCESS;
        }
        @catch (NSException* ex)
        {
            NSLog(@"%@", ex);
        }

        [communicator destroy];
    }
```

```
    return status;
}
```

There is quite a bit of code here, so let us examine each section in detail:

Objective-C

```
#import <objc/Ice.h>
#import <FileI.h>
#import <DirectoryI.h>
```

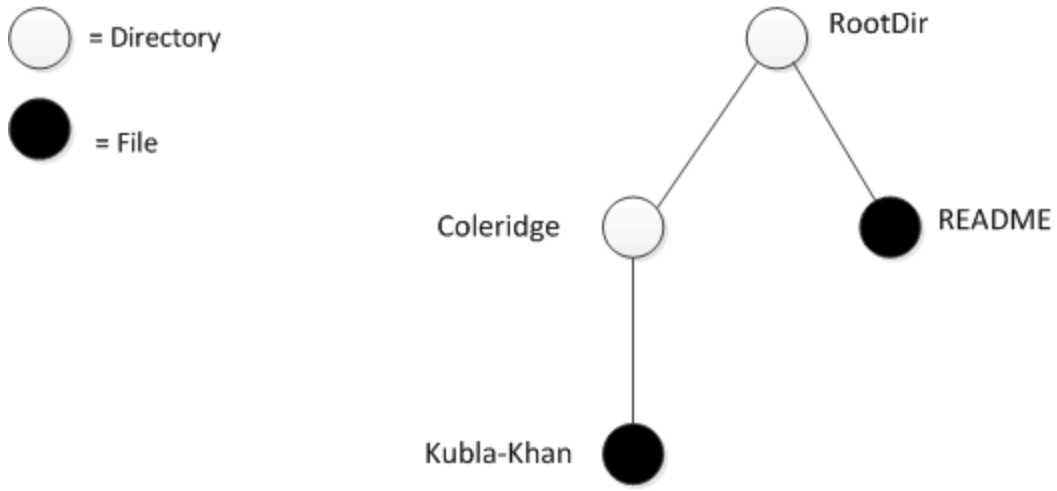
The code includes the header objc/`Ice.h`, which contains the definitions for the Ice run time, and the files `FileI.h` and `DirectoryI.h`, which contain the definitions of our servant implementations.

The next part of the source code is mostly boiler plate: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`, which blocks the calling thread until you call `shutdown` or `destroy` on the communicator. (Ice does not make any demands on the main thread, so `waitForShutdown` simply blocks the calling thread; if you want to use the main thread for other purposes, you are free to do so.)

Objective-C

```
int
main(int argc, char* argv[])
{
    int status = EXIT_FAILURE;
    @autoreleasepool
    {
        id<ICECommunicator> communicator = nil;
        @try
        {
            communicator = [ICEUtil createCommunicator:&argc argv:argv];

            id<ICEObjectAdapter> adapter = [communicator
createObjectAdapterWithEndpoints:@"SimpleFilesystem"

endpoints:@"default -p 10000"];



            // ...

            // All objects are created, allow client requests now
            //
            [adapter activate];

            //
            // Wait until we are done
            //
            [communicator waitForShutdown];
            status = EXIT_SUCCESS;
        }
        @catch (NSException* ex)
        {
            NSLog(@"%@", ex);
        }

        [communicator destroy];
    }
    return status;
}
```

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters: the name of the directory or file to be created and the servant for the parent directory. (For the root directory, which has no parent, we pass a `nil` parent.) Thus, the statement

Objective-C

```
DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in the above illustration shown:

Objective-C

```
        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
        [root activate:adapter];

        // Create a file called "README" in the root directory
        //
        FileI *file = [FileI filei:@"README" parent:root];
        NSMutableArray *text = [NSMutableArray arrayWithObject:
            @"This file system contains a collection of poetry."];
        [file write:text current:nil];
        [file activate:adapter];

        // Create a directory called "Coleridge" in the root dir
        //
        DirectoryI *coleridge =
[DirectoryI directoryi:@"Coleridge" parent:root];
        [coleridge activate:adapter];

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = [FileI filei:@"Kubla_Khan" parent:coleridge];
```

We first create the root directory and a file `README` within the root directory. (Note that we pass the servant for the root directory as the parent pointer when we create the new node of type `FileI`.)

After creating each servant, the code calls `activate` on the servant. (We will see the definition of this member function shortly.) The `activate` member function adds the servant to the ASM.

The next step is to fill the file with text:

Objective-C

```
        file = [FileI filei:@"Kubla_Khan" parent:coleridge];
        text = [NSMutableArray arrayWithObjects:
                            @"In Xanadu did Kubla Khan",
                            @"A stately pleasure-dome decree:",
                            @"Where Alph, the sacred river, ran",
                            @"Through caverns measureless to man",
                            @"Down to a sunless sea.",
                            nil];
        [file write:text current:nil];
        [file activate:adapter];
```

Recall that Slice sequences map to `NSArray` or `NSMutableArray`, depending on the parameter direction. Here, we instantiate that array and add a line of text to it.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

Objective-C

```
        [file write:text current:nil];
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via the pointer to the servant (of type `FileI`) and *not* via a proxy (of type `id<FilePrx>`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Objective-C function call. The operation implementation in the servant expects a `current` object. In this case, we pass nil (which is fine because the operation implementation does not use it anyway).

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the above illustration.

Servant Class Definitions in Objective-C

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the Objective-C classes `FileI` and `DirectoryI`. This means that our servant classes look as follows:

Objective-C

```
#import <Filesystem.h>

@interface FileI : FSFile <FSFile>
// ...
@end

@interface DirectoryI : FSDirectory <FSDirectory>
// ...
@end
```

Each servant class derives from its skeleton class and adopts its skeleton protocol.

We now can think about how to implement our servants. One thing that is common to all nodes is that they have a name and a parent directory. As we saw earlier, we pass these details to a convenience constructor, which also takes care of calling `autorelease` on the new servant.

In addition, we will use UUIDs as the object identities for files and directories. This relieves us of the need to otherwise come up with a unique identity for each servant (such as path names, which would only complicate our implementation). Because the `list` operation returns proxies to nodes, and because each proxy carries the identity of the servant it denotes, this means that our servants must store their own identity, so we can create proxies to them when clients ask for them.

For `File` servants, we also need to store the contents of the file, leading to the following definition for the `FileI` class:

Objective-C

```
#import <Filesystem.h>

@class DirectoryI;

@interface FileI : FSFile <FSFile>
{
    @private
        NSString *myName;
        DirectoryI *parent;
        ICEIdentity *ident;
        NSArray *lines;
}

@property(nonatomic, strong) NSString *myName;
@property(nonatomic, strong) DirectoryI *parent;
@property(nonatomic, strong) ICEIdentity *ident;
@property(nonatomic, strong) NSArray *lines;

+(id) filei:(NSString *)name parent:(DirectoryI *)parent;
-(void) write:(NSMutableArray *)text current:(ICECurrent *)current;
-(void) activate:(id<ICEObjectAdapter>)a;
@end
```

The instance variables store the name, parent node, identity, and the contents of the file. The `filei` convenience constructor instantiates the servant, remembers the name and parent directory, assigns a new identity, and calls `autorelease`.

Note that the only Slice operation we have defined here is the `write` method. This is necessary because, as we saw previously, the code in `Server.m` calls this method to initialize the files it creates.

For directories, the requirements are similar. They also need to store a name, parent directory, and object identity. Directories are also responsible for keeping track of the child nodes. We can store these nodes in an array of proxies. This leads to the following definition:

Objective-C

```
#import <Filesystem.h>

@interface DirectoryI : FSDirectory <FSDirectory>
{
    @private
        NSString *myName;
        DirectoryI *parent;
        ICEIdentity *ident;
        NSMutableArray *contents;
}
@property(nonatomic, strong) NSString *myName;
@property(nonatomic, strong) DirectoryI *parent;
@property(nonatomic, strong) ICEIdentity *ident;
@property(nonatomic, strong) NSMutableArray *contents;

+(id) directoryi:(NSString *)name parent:(DirectoryI *)parent;
-(void) addChild:(id<FSNodePrx>)child;
-(void) activate:(id<ICEObjectAdapter>)a;
@end
```

Because the code in `Server.m` does not call any Slice operations on directory servants, we have not declared any of the corresponding methods. (We will see the purpose of the `addChild` method shortly.) As for files, the convenience constructor creates the servant, remembers the name and parent, and assigns an object identity, as well as calling `autorelease`.

Servant Implementation in Objective-C

Let us now turn to how to implement each of the methods for our servants.

Implementing `FileI` in Objective-C

The implementation of the `name`, `read`, and `write` operations for files is trivial, returning or updating the corresponding instance variable:

Objective-C

```
-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(NSArray *) read:(ICECurrent *)current
{
    return lines;
}

-(void) write:(NSMutableArray *)text current:(ICECurrent *)current
{
    self.lines = text;
}
```

Note that this constitutes the complete implementation of the Slice operations for files.

Here is the convenience constructor:

Objective-C

```
+(id) filei:(NSString *)name parent:(DirectoryI *)parent
{
    FileI *instance = [[FileI alloc] init];
    if(instance == nil)
    {
        return nil;
    }
    instance.myName = name;
    instance.parent = parent;
    instance.ident = [ICEIdentity
identity:[ICEUtil generateUUID] category:nil];
    return instance;
}
```

After allocating and autoreleasing the instance, the constructor initializes the instance variables. The only interesting part of this code is how we create the identity for the servant. `generateUUID` is a class method of the `ICEUtil` class that returns a UUID. We assign this UUID to the `name` member of the identity.

We saw earlier that the server calls `activate` after it creates each servant. Here is the implementation of this method:

Objective-C

```
-(void) activate:(id<ICEObjectAdapter>)a
{
    id<FSNodePrx> thisNode =
[FSNodePrx uncheckedCast:[a add:self identity:ident]];
    [parent addChild:thisNode];
}
```

This is how our code informs the Ice run time of the existence of a new servant. The call to `add` on the object adapter adds the servant and object identity to the adapter's servant map. In other words, this step creates the link between the object identity (which is embedded in proxies), and the actual Objective-C class instance that provides the behavior for the Slice operations.

`add` returns a proxy to the servant, of type `id<ICEObjectPrx>`. Because the `contents` instance variable of directory servants stores proxies of type `id<FSNodePrx>` (and `addChild` expects a proxy of that type), we down-cast the returned proxy to `id<FSNodePrx>`. In this case, because we know that the servant we just added to the adapter is indeed a servant that implements the operations on the Slice `Node` interface, we can use an `uncheckedCast`.

The call to `addChild` connects the new file to its parent directory.

Implementing `DirectoryI` in Objective-C

The implementation of the Slice operations for directories is just as simple as for files:

Objective-C

```
-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(NSArray *) list:(ICECurrent *)current
{
    return contents;
}
```

Because the `contents` instance variable stores the proxies for child nodes of the directory, the `list` operation simply returns that variable.

The convenience constructor looks much like the one for file servants:

Objective-C

```
+(id) directoryi:(NSString *)name parent:(DirectoryI *)parent
{
    DirectoryI *instance = [[DirectoryI alloc] init];
    if(instance == nil)
    {
        return nil;
    }
    instance.myName = name;
    instance.parent = parent;
    instance.ident = [ICEIdentity
        identity:(parent ? [ICEUtil generateUUID] : @"RootDir")
        category:nil];
    instance.contents = [[NSMutableArray alloc] init];
    return instance;
}
```

The only noteworthy differences are that, for the root directory (which has no parent), the code uses `"RootDir"` as the identity. (As we saw earlier, the client knows that this is the identity of the root directory and uses it to create its proxy.)

The `addChild` method connects our nodes into a hierarchy by updating the `contents` instance variable. That way, each directory knows which nodes are contained in it:

Objective-C

```
-(void) addChild:(id<FSNodePrx>)child
{
     [contents addObject:child];
}
```

Finally, the `activate` is very much like the activate for files:

Objective-C

```
-(void) activate:(id<ICEObjectAdapter>)a
{
    id<FSNodePrx> thisNode = [FSNodePrx uncheckedCast:[a add:self
identity:ident]];
    [parent addChild:thisNode];
}
```

See Also

- Slice for a Simple File System
- Objective-C Mapping for Sequences
- Example of a File System Client in Objective-C
- The Ice Threading Model

Slice-to-Objective-C Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to Objective-C is identical to the mapping for these constructs without the `local` qualifier. The generated Objective-C code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

The rest of this section describes the mapping of the remaining local types to Objective-C:

Objective-C Mapping for Local Interface

On this page:

Mapped Objective-C Protocol

A Slice local interface is mapped to an Objective-C protocol with the same name, for example:

Slice

```
["objc:prefix:ICE"]
module Ice
{
    local interface Communicator
    {
        ...
    }
}
```

is mapped to the Objective-C protocol `ICECommunicator`:

Objective-C

```
@protocol ICECommunicator <NSObject>
...
@end
```

The `delegate` metadata allows you to map a local interface with a single operation to an Objective-C block. For example:

Slice

```
["objc:prefix:ICE"]
module Ice
{
    ["delegate"] local interface ValueFactory
    {
        Value create(string type);
    }
}
```

is mapped to:

Objective-C

```
typedef ICEObject* (^ICEValueFactory)(NSString*);
```

LocalObject in Objective-C

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to `ICELocalObject` in Objective-C:

Objective-C

```
@interface ICELocalObject : NSObject
{
   // internal member
}
@end
```

Mapping for Local Interface Inheritance in Objective-C

Inheritance of local Slice interfaces is mapped to protocol inheritance in Objective-C. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

Objective-C

```
@protocol MA <NSObject>
@end

@protocol MB <MA>
@end

@protocol MC <MA>
@end

@protocol MD <MB, MC>
@end
```

Objective-C Mapping for Local Classes

On this page:

Mapped Objective-C Class

A local Slice class is mapped to an Objective-C protocol and interface with the same name. For example:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ...
    }
}
```

is mapped to the Objective-C protocol and interface `ICEConnectionInfo`:

Objective-C

```
@protocol ICEConnectionInfo <NSObject>
@end

@interface ICEConnectionInfo : ICELocalObject
...
@end
```

LocalObject in Objective-C

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to `ICELocalObject` in Objective-C.

Mapping for Local Interface Inheritance in Objective-C

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` and `implements` a re mapped to interface and protocol inheritance in Objective-C. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

Objective-C

```
@protocol MA <NSObject>
@end

@protocol MB <NSObject>
@end

@protocol MC <MA, MB>
@end

@interface MC : ICELocalObject
+(id) c;
@end

@protocol MD <MC>
@end

@interface MD : MC
+(id) d;
@end
```

Objective-C Mapping for Local Exceptions

On this page:

Mapped Objective-C Interface

A local Slice exception is mapped to an Objective-C interface with the same name. For example:

Slice

```
["objc:prefix:ICE"]
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the Objective-C interface `ICEInitializationException`:

Objective-C

```
@interface ICEInitializationException : ICELocalException
...
@end
```

Base Interface for Local Exceptions in Objective-C

All mapped Objective-C interfaces derive directly or indirectly from `ICELocalException`:

Objective-C

```
@interface ICELocalException : ICEException
{
@protected
    const char* file;
    int line;
}

@property(nonatomic, readonly) NSString* file;
@property(nonatomic, readonly) int line;

-(id)init:(const char*)file line:(int)line;
-(id)init:(const char*)file line:(int)line reason:(NSString*)reason;
+(id)localException:(const char*)file line:(int)line;
@end
```

Mapping for Local Exception Inheritance in Objective-C

A local Slice exception can extend another Slice exception. In Objective-C, this is mapped to inheritance of the corresponding interfaces. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

Objective-C

```
@interface MErrorBase : ICELocalException
-(NSString *) ice_id;
+(id) errorBase:(const char*)file line:(int)line;
@end

@interface MResourceError : MErrorBase
-(NSString *) ice_id;
+(id) resourceError:(const char*)file line:(int)line;
@end
```

Objective-C Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to an Objective-C method with the same name. The mapping of operation parameters to Objective-C is identical to the Client-Side Mapping for these parameters. However, unlike the Client-Side mapping, there is no overloaded method with a trailing `context` parameter.

The type of a parameter can be a local interface or class. Such a parameter is passed as an `id<`*`mapped Objective-C protocol`*`>` for regular parameters; a "delegate" interface parameter, mapped to a block in Objective-C, is passed as this block.

A `LocalObject` parameter is mapped to a parameter of type `id`. You can also create constructed types (such as local sequences and local dictionaries) with local types.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

Objective-C

```
typedef NSArray MLSeq;
typedef NSMutableArray MMutableLSeq;

@protocol ML <NSObject>
-(NSMutableString*) op:(ICEInt)n s:(NSString*)s any:(id)any m:(ICEInt*)m
t:(NSMutableString**)t newLSeq:(MMutableLSeq**)newLSeq;
@end
```

Objective-C Mapping for Data Members in Local Types

Data members on local Slice types (classes, exceptions and structs) are mapped to Objective-C just like the data members of the corresponding non local Slice construct.

A local Slice type can have a data member of type local interface or class, which is mapped to an Objective-C property of type `ICELocalObject<`*`mapped Objective-C protocol`*`>*`. A `LocalObject` data member is mapped to an Objective-C property of the same name with type `id`.

PHP Mapping

Ice currently provides a client-side mapping for PHP, but not a server-side mapping.

Topics

Client-Side Slice-to-PHP Mapping

The client-side Slice-to-PHP mapping defines how Slice data types are translated to PHP types, and how clients invoke operations, pass parameters, and handle errors. Much of the PHP mapping is intuitive. For example, Slice sequences map to PHP arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in PHP.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the PHP mapping, you should need no more than the Slice definition of your application and knowledge of the PHP mapping rules. In particular, looking through the generated code in order to discern how to use the PHP mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

Slice to PHP mapping supports two mappings, the namespace mapping that maps Slices modules to PHP namespaces is the default with Ice 3.7, the flattened mapping is now deprecated.

> **The `Ice` Module**
> All of the APIs for the Ice run time are nested in the `Ice` module, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` module are generated from Slice definitions; other parts of the `Ice` module provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` m odule throughout the remainder of the manual.
>
> A PHP application can load the Ice run time using the `require` statement:
>
> ```
> require 'Ice.php';
> ```
>
> If the statement executes without error, the Ice run time is loaded and available for use. You can determine the version of the Ice run time you have just loaded by calling the `stringVersion` function:
>
> ```
> $icever = \Ice\stringVersion();
> ```

Topics

PHP Mapping for Identifiers

A Slice identifier maps to an identical PHP identifier. For example, the Slice identifier `Clock` becomes the PHP identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a PHP keyword or is an identifier reserved by the Ice run time (such as `checked Cast`), the corresponding PHP identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.

> You should try to avoid such identifiers as much as possible.

A single Slice identifier often results in several PHP identifiers. For example, for a Slice interface named `Foo`, the generated PHP code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_while` and `whilePrx` (*not* `_whilePrx`), that is, the underscore prefix is applied only to those generated identifiers that actually require it.

See Also

- Lexical Rules
- PHP Mapping for Modules
- PHP Mapping for Built-In Types
- PHP Mapping for Enumerations
- PHP Mapping for Structures
- PHP Mapping for Sequences
- PHP Mapping for Dictionaries
- PHP Mapping for Constants
- PHP Mapping for Exceptions

PHP Mapping for Modules

By default a Slice modules map to PHP namespaces with the same name as the Slice module. Consider the following Slice definition:

Slice

```
module M
{
    module N
    {
        enum Color { red, green, blue }
    }
}
```

The Slice identifier `Color` maps to `\M\N\Color`.

See Also

- Modules
- PHP Mapping for Identifiers
- PHP Mapping for Built-In Types
- PHP Mapping for Enumerations
- PHP Mapping for Structures
- PHP Mapping for Sequences
- PHP Mapping for Dictionaries
- PHP Mapping for Constants
- PHP Mapping for Exceptions

PHP Mapping for Built-In Types

On this page:

Mapping of Slice Built-In Types to PHP Types

PHP has a limited set of primitive types: `boolean`, `integer`, `double`, and `string`. The Slice built-in types are mapped to PHP types as shown in the table below:

Slice	Ruby
`bool`	`true` or `false`
`byte`	`integer`
`short`	`integer`
`int`	`integer`
`long`	`integer`
`float`	`double`
`double`	`double`
`string`	`string`

PHP's `integer` type may not accommodate the range of values supported by Slice's `long` type, therefore `long` values that are outside this range are mapped as strings. Scripts must be prepared to receive an integer or string from any operation that returns a `long` value.

String Mapping in PHP

String values returned as the result of a Slice operation (including return values, out parameters, and data members) contain UTF-8 encoded strings unless the program has installed a string converter, in which case string values use the converter's native encoding instead.

As string input values for a remote Slice operation, Ice accepts `null` in addition to `string` objects; each occurrence of `null` is marshaled as an empty string. Ice assumes that all `string` objects contain valid UTF-8 encoded strings unless the program has installed a string converter, in which case Ice assumes that `string` objects use the native encoding expected by the converter.

See Also

- Basic Types
- PHP Mapping for Identifiers
- PHP Mapping for Modules
- PHP Mapping for Enumerations
- PHP Mapping for Structures
- PHP Mapping for Sequences
- PHP Mapping for Dictionaries
- PHP Mapping for Constants
- PHP Mapping for Exceptions
- C++98 Strings and Character Encoding

PHP Mapping for Enumerations

PHP does not have an enumerated type, so a Slice enumeration is mapped to a PHP class: the name of the Slice enumeration becomes the name of the PHP class; for each enumerator, the class contains a constant with the same name as the enumerator. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The generated PHP class looks as follows:

PHP

```
class Fruit
{
    const Apple = 0;
    const Pear = 1;
    const Orange = 2;
}
```

Suppose we modify the Slice definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

The generated PHP class changes accordingly:

PHP

```
class Fruit
{
    const Apple = 0;
    const Pear = 3;
    const Orange = 4;
}
```

Since enumerated values are mapped to integer constants, application code is not required to use the generated constants. When an enumerated value enters the Ice run time, Ice validates that the given integer is a valid value for the enumeration. However, to minimize the potential for defects in your code, we recommend using the generated constants instead of literal integers.

See Also

- Enumerations
- PHP Mapping for Identifiers
- PHP Mapping for Modules
- PHP Mapping for Built-In Types
- PHP Mapping for Structures
- PHP Mapping for Sequences
- PHP Mapping for Dictionaries
- PHP Mapping for Constants
- PHP Mapping for Exceptions

PHP Mapping for Structures

A Slice structure maps to a PHP class containing a public variable for each member of the structure. For example, here is our `Employee` structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The PHP mapping generates the following definition for this structure:

PHP

```
class Employee
{
    public function __construct($number=0, $firstName='', $lastName='');

    public function __toString();

    public $number;
    public $firstName;
    public $lastName;
}
```

The class provides a constructor whose arguments correspond to the data members. This allows you to instantiate and initialize the class in a single statement (instead of having to first instantiate the class and then assign to its members). Each argument provides a default value appropriate for the member's type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

You can also declare different default values for members of primitive and enumerated types.

The mapping also includes a definition for the `__toString` magic method, which returns a string representation of the structure.

See Also

PHP Mapping for Sequences

A Slice sequence maps to a native PHP indexed array. The first element of the Slice sequence is contained at index 0 (zero) of the PHP array, followed by the remaining elements in ascending index order.

Consider this example:

Slice

```
sequence<Fruit> FruitPlatter;
```

You can create an instance of `FruitPlatter` as shown below:

PHP

```
// Make a small platter with one Apple and one Orange
//
$platter = array(Fruit::Apple, Fruit::Orange);
```

The Ice run time validates the elements of an array to ensure that they are compatible with the declared type and raises `InvalidArgumentException` if an incompatible type is encountered.

See Also

- Sequences
- PHP Mapping for Identifiers
- PHP Mapping for Modules
- PHP Mapping for Built-In Types
- PHP Mapping for Enumerations
- PHP Mapping for Structures
- PHP Mapping for Dictionaries
- PHP Mapping for Constants
- PHP Mapping for Exceptions

PHP Mapping for Dictionaries

A Slice dictionary maps to a native PHP associative array. The PHP mapping does not currently support all Slice dictionary types, however, because native PHP associative arrays support only integers and strings as keys.

A Slice dictionary whose key type is an enumeration or one of the primitive types `boolean`, `byte`, `short`, `int`, or `long` is mapped as an associative array with an integer key.

> Boolean values are treated as integers, with false equivalent to 0 (zero) and true equivalent to 1 (one).

A Slice dictionary with a `string` key type is mapped as an associative array with a string key. All other key types cause a warning to be generated.

Here is the definition of our `EmployeeMap`:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

You can create an instance of this dictionary as shown below:

PHP

```
$e1 = new Employee;
$e1->number = 42;
$e1->firstName = "Stan";
$e1->lastName = "Lipmann";

$e2 = new Employee;
$e2->number = 77;
$e2->firstName = "Herb";
$e2->lastName = "Sutter";

$em = array($e1->number => $e1, $e2->number => $e2);
```

The Ice run time validates the elements of a dictionary to ensure that they are compatible with the declared type; `InvalidArgumentException` exception is raised if an incompatible type is encountered.

See Also

- Dictionaries
- PHP Mapping for Identifiers
- PHP Mapping for Modules
- PHP Mapping for Built-In Types
- PHP Mapping for Enumerations
- PHP Mapping for Structures
- PHP Mapping for Sequences
- PHP Mapping for Constants
- PHP Mapping for Exceptions

PHP Mapping for Constants

A Slice constant maps to a PHP constant. Consider the following definitions:

Slice

```
module M
{
    const bool      AppendByDefault = true;
    const byte      LowerNibble = 0x0f;
    const string    Advice = "Don't Panic!";
    const short     TheAnswer = 42;
    const double    PI = 3.1416;

    enum Fruit { Apple, Pear, Orange };
    const Fruit     FavoriteFruit = Pear;
}
```

The mapping for these constants is shown below:

PHP

```
namespace M
{
    define(__NAMESPACE__ . '\\AppendByDefault', true);
    define(__NAMESPACE__ . '\\LowerNibble', 15);
    define(__NAMESPACE__ . '\\Advice', "Don't Panic!");
    define(__NAMESPACE__ . '\\TheAnswer', 42);
    define(__NAMESPACE__ . '\\PI', 3.1416);
    define(__NAMESPACE__ . '\\FavoriteFruit', \M\Fruit::Pear);
}
```

Slice string literals that contain non-ASCII characters or universal character names are mapped to PHP string literals with these characters replaced by their UTF-8 encoding as octal escapes. For example:

Slice

```
module M
{
    const string Egg = "œuf";
    const string Heart = "c\u0153ur";
    const string Banana = "\U0001F34C";
}
```

is mapped to:

PHP

```
namespace M
{
    define(__NAMESPACE__ . '\\Egg', "\305\223uf");
    define(__NAMESPACE__ . '\\Heart', "c\305\223ur");
    define(__NAMESPACE__ . '\\Banana', "\360\237\215\214");
}
```

Slice constants are mapped to PHP constants in the enclosing namespace:

PHP

```
$ans = \M\TheAnswer;
```

See Also

- Constants and Literals
- PHP Mapping for Identifiers
- PHP Mapping for Modules
- PHP Mapping for Built-In Types
- PHP Mapping for Enumerations
- PHP Mapping for Structures
- PHP Mapping for Sequences
- PHP Mapping for Dictionaries
- PHP Mapping for Exceptions

PHP Mapping for Exceptions

On this page:

Inheritance Hierarchy for Exceptions in PHP

The ancestor of all exceptions is `Exception`, from which `Ice\Exception` is derived. `Ice\LocalException` and `Ice\UserException` are derived from `Ice\Exception` and form the base for all run-time and user exceptions.

PHP Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map to the abbreviated PHP class definitions shown below:

PHP

```
class GenericError extends \Ice\UserException
{
    public function __construct($reason='');
    public function ice_id();
    public function __toString();

    public $reason;
}

class BadTimeVal extends GenericError
{
    public function __construct($reason='');
    public function ice_id();
    public function __toString();
}

class BadZoneName extends GenericError
{
    public function __construct($reason='');
    public function ice_id();
    public function __toString();
}
```

Each Slice exception is mapped to a PHP class with the same name. The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception member corresponds to an instance variable of the instance, which the constructor initializes to a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

You can also declare different default values for members of primitive and enumerated types. For derived exceptions, the constructor has one parameter for each of the base exception's data members, plus one parameter for each of the derived exception's data members, in base-to-derived order. As an example, although `BadTimeVal` and `BadZoneName` do not declare data members, their constructors still accept a value for the inherited data member `reason` in order to pass it to the constructor of the base exception `GenericError`.

Each exception also defines the `ice_id` function that returns the Slice type ID of the exception, as well as the `__toString` magic function to return a stringified representation of the exception and its members.

All user exceptions are derived from the base class `Ice\UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice\UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice\LocalException`, and you can catch all Ice exceptions with a handler for `Ice\Exception`.

Optional Data Members

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `\Ice\None` to indicate that the member is unset. A well-behaved program must compare an optional data member to `\Ice\None` before using the member's value:

PHP

```
try
{
    ...
}
catch(\Ice\Exception $ex)
{
    if($ex->optionalMember == \Ice\None)
    {
        echo "optionalMember is unset\n";
    }
    else
    {
        echo "optionalMember = " . $ex->optionalMember . "\n";
    }
}
```

The `\Ice\None` marker value has different semantics than `null`. Since `null` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to `null` is considered to be set. If you need to distinguish between an unset value and a value set to `null`, you can do so as follows:

PHP

```
try
{
    ...
}
catch(\Ice\Exception $ex)
{
    if($ex->optionalMember == \Ice\None)
    {
        echo "optionalMember is unset\n";
    }
    else if($ex->optionalMember == null)
    {
        echo "optionalMember is null\n";
    }
    else
    {
        echo "optionalMember = " . $ex->optionalMember . "\n";
    }
}
```

PHP Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `\Ice\LocalException` (which, in turn, derives from `\Ice\Exception`).

By catching exceptions at the appropriate point in the inheritance hierarchy, you can handle exceptions according to the category of error they indicate:

- `\Ice\LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `\Ice\UserException`
  This is the root of the inheritance tree for user exceptions.
- `\Ice\TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `\Ice\ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, `\Ice\ConnectTimeoutException` can be handled as `\Ice\ConnectTimeoutException`, `\Ice\TimeoutException`, `\Ice\LocalException`, or `\Ice\Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `\Ice\LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `\Ice\FacetNotExistException` and `\Ice\ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- PHP Mapping for Identifiers
- PHP Mapping for Modules
- PHP Mapping for Built-In Types

PHP Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Objects in PHP

Slice interfaces are implemented by instances of the `\Ice\ObjectPrx` class. In the client's address space, an instance of `ObjectPrx` is the local ambassador for a remote instance of an interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

The PHP mapping for proxies differs from that of other Ice language mappings in that the `ObjectPrx` class is used to implement *all* Slice interfaces. The primary motivation for this design is minimizing the amount of code that is generated for each interface. As a result, a proxy object returned by the communicator operations `stringToProxy` and `propertyToProxy` is *untyped*, meaning it is not associated with a user-defined Slice interface. Once you narrow the proxy to a particular interface, you can use that proxy to invoke your Slice operations.

Proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

For each operation in the interface, the proxy object supports a method of the same name. Each operation accepts an optional trailing parameter representing the operation context. This parameter is an associative string array for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the context parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Interface Inheritance in PHP

Consider the following example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

Given a proxy that has been narrowed to `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

`Ice\ObjectPrx` Class in PHP

In the PHP language mapping, all proxies are instances of `Ice\ObjectPrx`. This class provides a number of methods:

PHP

```
namespace Ice
{
    class ObjectPrx
    {
        function ice_getIdentity();
        function ice_isA($id);
        function ice_ids();
        function ice_id();
        function ice_ping();
        # ...
    }
}
```

The methods behave as follows:

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

  ```
  module Ice
  {
      struct Identity
      {
          string name;
          string category;
      }
  }
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

  PHP

  ```
  $proxy1 = ...
  $proxy2 = ...
  $id1 = $proxy1->ice_getIdentity();
  $id2 = $proxy2->ice_getIdentity();

  if($id1 == $id2)
  {
      // proxy1 and proxy2 denote the same object
  }
  else
  {
      // proxy1 and proxy2 denote different objects
  }
  ```

- **ice_isA**
  The ice_isA method determines whether the object denoted by the proxy supports a specific interface. The argument to ice_isA is a type ID. For example, to see whether a proxy of type ObjectPrx denotes a Printer object, we can write:

  PHP

  ```
  $proxy = ...
  if($proxy != null && $proxy->ice_isA("::Printer"))
  {
      // proxy denotes a Printer object
  }
  else
  {
      // proxy denotes some other type of object
  }
  ```

  Note that we are testing whether the proxy is null before attempting to invoke the ice_isA method. This avoids getting a run-time error if the proxy is null.

- **ice_ids**
  The ice_ids method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- **ice_id**
  The ice_id method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type BasePrx, with a static type ID of ::Base, the return value of ice_id might be "::Base", or it might be something more derived, such as "::Derived".

- **ice_ping**
  The ice_ping method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

The ObjectPrx class also defines an operator for comparing two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, == returns false even though the proxies denote the same object.

The ice_isA, ice_ids, ice_id, and ice_ping methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in ObjectPrx, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Proxy Helper Classes in PHP

The PHP mapping for a proxy generates a helper class with several static methods. For example, the following class is generated for the Slice interface named Simple:

PHP

```
class SimplePrxHelper
{
    public static function checkedCast($proxy, $facetOrCtx=null,
$ctx=null);

    public static function uncheckedCast($proxy, $facet=null);

    public static function ice_staticId();
}
```

The `checkedCast` and `uncheckedCast` methods are described in the following section.

The `ice_staticId` method returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Casting Proxies in PHP

As shown above, the proxy's helper class includes two static methods that support down-casting:

PHP

```
class SimplePrxHelper
{
    public static function checkedCast($proxy, $facetOrCtx=null,
$ctx=null);
    public static function uncheckedCast($proxy, $facet=null);

    // ...
}
```

The method names `checkedCast` and `uncheckedCast` are reserved for use in proxies. If a Slice interface defines an operation with either of those names, the mapping escapes the name in the generated proxy by prepending an underscore. For example, an interface that defines an operation named `checkedCast` is mapped to a proxy with a method named `_checkedCast`.

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a proxy narrowed to that type; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is `null`), the cast returns `null`.

The arguments are described below:

- `$proxy`
  The proxy to be narrowed.
- `$facetOrCtx`
  This optional argument can be either a string representing a desired facet, or an associative string array representing a context.
- `$ctx`
  If `$facetOrCtx` contains a facet name, use this argument to supply an associative string array representing a context.
- `$facet`
  Specifies the name of the desired facet.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

PHP

```
$obj = ...      // Get a proxy from somewhere...

$simple = SimplePrxHelper::checkedCast($obj);
if($simple != null)
{
    // Object supports the Simple interface...
}
else
{
    // Object is not of type Simple...
}
```

Note that a `checkedCast` contacts the server. This is necessary because only the server implementation has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

PHP

```
$obj = ...                                          // Get proxy...
$process = ProcessPrxHelper::uncheckedCast($obj); // No worries...
$process->launch(40, 60);                           // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

Proxy Backward Compatibility in PHP

Prior releases of the PHP language mapping provided two proxy methods for narrowing a proxy:

PHP

```
namespace Ice
{
    class ObjectPrx
    {
        function ice_checkedCast($type, $facetOrCtx=null, $ctx=null);
        function ice_uncheckedCast($type, $facet=null);
        # ...
    }
}
```

For example, a proxy can be narrowed as follows:

PHP

```
$proxy = $proxy->ice_checkedCast("::Demo::Hello");
```

Embedding such type ID strings in your application is a potential source of defects because the strings are not validated until run time. Although these methods are still supported for the sake of backward compatibility, we recommend using the static methods that are generated in the helper class corresponding to each interface, as shown below:

PHP

```
$proxy = \Demo\HelloPrxHelper::checkedCast($proxy);
```

Not only are these static methods consistent with the APIs of other Ice language mappings, they also avoid the need to hard-code type ID strings in your application.

Using Proxy Methods in PHP

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

PHP

```
$proxy = $communicator->stringToProxy(...);
$proxy = $proxy->ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to

repeat a down-cast after using a factory method. The example below demonstrates these semantics:

PHP

```
$base = $communicator->stringToProxy(...);
$hello = \Demo\HelloPrxHelper::checkedCast($base);
$hello = $hello->ice_invocationTimeout(10000); // Type is not discarded
$hello->sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return an untyped proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in PHP

Proxy objects support comparison using the comparison operators == and `!=`. Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

PHP

```
$p1 = ...        // Get a proxy...
$p2 = ...        // Get another proxy...

if($p1 != $p2)
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each uses a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` module:

PHP

```
namespace Ice
{
    function proxyIdentityCompare($lhs, $rhs);
    function proxyIdentityAndFacetCompare($lhs, $rhs);
}
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

PHP

```
$p1 = ...        // Get a proxy...
$p2 = ...        // Get another proxy...

if(\Ice\proxyIdentityCompare($p1, $p2) != 0)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, `-1` if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major sort key and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Type IDs
- PHP Mapping for Operations
- Request Contexts
- Versioning
- IceStorm

PHP Mapping for Operations

On this page:

Basic PHP Mapping for Operations

As we saw in the PHP mapping for interfaces, for each operation on an interface, a proxy object narrowed to that type supports a corresponding method with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

PHP

```
$node = ...             // Initialize proxy
$name = $node->name();  // Get name via RPC
```

Normal and `idempotent` Operations in PHP

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect.

Passing Parameters in PHP

In-Parameters in PHP

The PHP mapping for `in` parameters guarantees that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

A proxy object narrowed to the `ClientToServer` interface supports the following methods:

PHP

```
function op1($i, $f, $b, $s, $context=null);
function op2($ns, $ss, $st, $context=null);
function op3($proxy, $context=null);
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

PHP

```
$p = ...                                    // Get proxy...

$p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

$i = 42;
$f = 3.14;
$b = true;
$s = "Hello world!";
$p->op1($i, $f, $b, $s);                // Pass simple variables

$ns = new NumberAndString;
$ns->x = 42;
$ns->str = "The Answer";
$ss = array("Hello world!");
$st = array();
$st[0] = $ns;
$p->op2($ns, $ss, $st);                 // Pass complex variables

$p->op3($p);                            // Pass proxy
```

Out-Parameters in PHP

Out parameters are passed by reference. Here is the same Slice definition we saw earlier, but this time with all parameters being passed in the out direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    int op1(out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The PHP mapping looks the same as it did for the in-parameters version:

PHP

```
function op1($i, $f, $b, $s, $context=null);
function op2($ns, $ss, $st, $context=null);
function op3($proxy, $context=null);
```

Given a proxy to a `ServerToClient` interface, the client code can receive the results as in the following example:

PHP

```
$p = ...                  // Get proxy...
$p->op1($i, $f, $b, $s);
$p->op2($ns, $ss, $st);
$p->op3($stcp);
```

Note that it is not necessary to use the reference operator (&) before each argument because the Ice run time forces each `out` parameter to have reference semantics.

Parameter Type Mismatches in PHP

The Ice run time performs validation on the arguments to a proxy invocation and reports any type mismatches as `InvalidArgumentException`.

Null Parameters in PHP

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, it makes no difference to the receiver whether you send a string as `null` or as an empty string: either way, the receiver sees an empty string.

Optional Parameters in PHP

Optional parameters use the same mapping as required parameters. The only difference is that `\Ice\None` can be passed as the value of an optional parameter or return value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

A client can invoke this operation as shown below:

PHP

```
$i = $proxy->execute("--file log.txt", $v);
$i = $proxy->execute(\Ice\None, $v);

if($v != Ice_Unset)
{
    echo "value = " . $v . "\n";
}
```

A well-behaved program must always compare an optional parameter to `\Ice\None` prior to using its value. Keep in mind that the `\Ice\None` marker value has different semantics than `null`. Since `null` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional parameter is set. An optional parameter set to `null` is considered to be set. If you need to distinguish between an unset parameter and a parameter set to `null`, you can do so as follows:

PHP

```
if($optionalParam == \Ice\None)
{
    echo "optionalParam is unset\n";
}
else if($optionalParam == null)
{
    echo "optionalParam is null\n";
}
else
{
    echo "optionalParam = " . $optionalParam . "\n";
}
```

Exception Handling in PHP

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as PHP exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

PHP

```
$child = ...        // Get child proxy...

try
{
    $child->askToCleanUp();
}
catch(Tantrum $t)
{
    echo "The child says: " . $t->reason . "\n";
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will usually be handled by exception handlers higher in the hierarchy. For example:

PHP

```
try
{
    $child = ...          // Get child proxy...
    try
    {
        $child->askToCleanUp();
        $child->praise(); // Give positive feedback...
    }
    catch(Tantrum $t)
    {
        echo "The child says: " . $t->reason . "\n";
        $child->scold();  // Recover from error...
    }
}
catch(\Ice\LocalException $ex)
{
    echo $ex->__toString() . "\n";
}
```

See Also

- Operations
- Hello World Application
- Slice for a Simple File System
- PHP Mapping for Interfaces
- PHP Mapping for Exceptions

PHP Mapping for Classes

On this page:

Basic PHP Mapping for Classes

A Slice class maps to a PHP class with the same name. For each Slice data member, the generated class contains a member variable, just as for structures and exceptions. Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

The PHP mapping generates the following code for this definition:

PHP

```
class TimeOfDay extends \Ice\Value
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    public function ice_id()
    {
        return '::TimeOfDay';
    }

    public static function ice_staticId()
    {
        return '::TimeOfDay';
    }

    public function __toString()
    {
        // ...
    }

    public $hour;
    public $minute;
    public $second;
}
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `\Ice\Value`. This reflects the semantics of Slice classes in that all classes implicitly inherit from `\Ice\Value`, which is the ultimate ancestor of all classes. Note that `\Ice\Value` is *not* the same as `\Ice\ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The constructor initializes an instance variable for each Slice data member.
3. The class defines the method `ice_id` and class method `ice_staticId`.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Value` in PHP

Like interfaces, classes implicitly inherit from a common base class, `\Ice\Value`. However, classes inherit from `\Ice\Value` instead of `\Ice\ObjectPrx`, therefore you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`Value` defines the following functions:

PHP

```
class Value
{
    public function ice_id()
    {
        return "::Ice::Object";
    }

    public function ice_preMarshal()
    {
    }

    public function ice_postMarshal()
    {
    }

 public function ice_getSlicedData()
    {
        ...
    }

    public static function ice_staticId()
    {
        return "::Ice::Object";
    }
}
```

These functions behave as follows:

- `ice_id`
  This method returns the actual run-time type ID of the object. If you call `ice_id` through a reference to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.
- `ice_preMarshal`
  If the object defines this method, the Ice run time invokes it just prior to marshaling the object's state, providing the opportunity for the object to validate its declared data members.
- `ice_postUnmarshal`
  If the object defines this method, the Ice run time invokes it after unmarshaling the object's state. An object typically defines this method when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.
- `ice_staticId`
  This method is generated in each class and returns the static type ID of the class.

Class Data Members in PHP

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding member variable.

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `\Ice\Unset` to indicate that the member is unset. A well-behaved program must compare an optional data member to `Unset` before using the member's value:

PHP

```
$v = ...;
if($v->optionalMember == \Ice\None)
{
    echo "optionalMember is unset\n";
}
else
{
    echo "optionalMember = " . $v->optionalMember . "\n";
}
```

The `Unset` marker value has different semantics than `null`. Since `null` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to `null` is considered to be set. If you need to distinguish between an unset value and a value set to `null`, you can do so as follows:

PHP

```
$v = ...;
if($v->optionalMember == \Ice\None)
{
    echo "optionalMember is unset\n";
}
else if($v->optionalMember == null)
{
    echo "optionalMember is null\n";
}
else
{
    echo "optionalMember = " . $v->optionalMember . "\n";
}
```

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
}
```

The Slice compiler produces the following generated code for this definition:

PHP

```
abstract class TimeOfDay extends \Ice\Value
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    public function ice_id()
    {
        return '::TimeOfDay';
    }

    public static function ice_staticId()
    {
        return '::TimeOfDay';
    }

    public function __toString()
    {
        // ...
    }

    protected $hour;
    protected $minute;
    protected $second;
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

Class Constructors in PHP

Classes have a constructor that assigns to each data member a default value appropriate for its type:

Data Member Type	Default Value

`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

You can also declare different default values for data members of primitive and enumerated types.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

Pass the marker value `\Ice\None` as the value of any optional data members that you wish to be unset.

Class Operations in PHP

> **Deprecated Feature**
> Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

With the PHP mapping, operations in classes are not mapped at all into the corresponding PHP class. The generated PHP class is the same whether the Slice class has operations or not.

Value Factories in PHP

> While value factories were necessary in previous versions of Ice when using classes with operations (a now deprecated feature) with the PHP mapping, value factories may be used for any kind of class and are *not* deprecated.

Value factories allow you to create classes derived from the PHP class generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
class CustomTimeOfDay extends TimeOfDay
{
    public function format() { ... prints formatted data members ... }
}
```

You then create and register a value factory for your custom class with your Ice communicator:

PHP

```
class ValueFactory implements \Ice\ValueFactory
{
    public function create($type)
    {
        if($type == TimeOfDay::ice_staticId()))
        {
            return new CustomTimeOfDay;
        }
        assert(false);
        return null;
    }
}

$communicator->getValueFactoryManager()->add(new ValueFactory(),
TimeOfDay::ice_staticId())
```

See Also

- Classes
- Type IDs
- Optional Data Members
- Value Factories

slice2php Command-Line Options

On this page:

`slice2php` Command-Line Options

The Slice-to-PHP compiler, `slice2php`, offers the following command-line options in addition to the standard options:

- `--all`
  Generate code for all Slice definitions, including those included by the main Slice file.
- `--no-namespace`
  Generate code without support for PHP namespaces (deprecated).
- `--checksum`
  Generate checksums for Slice definitions.

Compiler Output in PHP

For each Slice file `X.ice`, `slice2php` generates PHP code into a file named `X.php` in the output directory. The default output directory is the current working directory, but a different directory can be specified using the `--output-dir` option.

Include Files in PHP

It is important to understand how `slice2php` handles include files. In the absence of the `--all` option, the compiler does not generate PHP code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into PHP `require` statements in the following manner:

1. Determine the full pathname of the included file.
2. Create the shortest possible relative pathname for the included file by iterating over each of the include directories (specified using the `-I` option) and removing the leading directory from the included file if possible.
   For example, if the full pathname of an included file is `/opt/App/slice/OS/Process.ice`, and we specified the options `-I/opt/App` and `-I/opt/App/slice`, then the shortest relative pathname is `OS/Process.ice` after removing `/opt/App/slice`.
3. Replace the `.ice` extension with `.php`. Continuing our example from the previous step, the translated `require` statement becomes

   ```
   require_once "OS/Process.php";
   ```

   As a result, you can use `-I` options to tailor the `require` statements generated by the compiler in order to avoid absolute path names and match the organizational structure of your application's source files.

See Also

- Using the Slice Compilers

Application Notes for PHP

On this page:

PHP Request Semantics

In PHP terminology, a *request* is the execution of a PHP script on behalf of a Web client. Each request essentially runs in its own instance of the PHP interpreter, isolated from any other requests that may be executing concurrently. Upon the completion of a request, the interpreter reclaims memory and other resources that were acquired during the request, including objects created by the Ice extension.

Using Communicators in PHP

A communicator represents an instance of the Ice run time. A PHP script that needs to invoke an operation on a remote Ice object must initialize a communicator, obtain and narrow a proxy, and make the invocation. For example, here is a minimal (but complete) Ice script:

PHP

```
<?php
require_once 'Ice.php';
require_once 'Hello.php';

$communicator = null;

try
{
    $data = new Ice\InitializationData;
    $data->properties = Ice\createProperties();
    $data->properties->load("props.cfg");
    $communicator = Ice\initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo\HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice\LocalException $ex)
{
    // Deal with exception...
}

if($communicator)
{
    try
    {
        $communicator->destroy();
    }
    catch(Ice\LocalException $ex)
    {
        // Ignore.
    }
}
?>
```

By default, the Ice extension automatically destroys any communicator that was created during a request. This means a script can usually omit the call to `destroy` unless there is an application-specific reason to destroy the communicator explicitly. Consequently, we can simplify our script to the following:

PHP

```
<?php
require 'Ice.php';
require 'Hello.php';

try
{
    $data = new Ice\InitializationData;
    $data->properties = Ice\createProperties();
    $data->properties->load("props.cfg");
    $communicator = Ice\initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo\HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice\LocalException $ex)
{
    // Deal with exception...
}
?>
```

Now we allow the Ice extension to destroy our communicator automatically. (The extension traps and ignores any exception raised by `destroy`.)

Although the automatic destruction of communicators is convenient, it is important to consider the performance characteristics of this script. Specifically, each execution of the script involves the following activities:

1. Create an Ice property set
2. Load and parse a property file
3. Initialize a communicator with the given configuration properties
4. Obtain a proxy for the remote Ice object
5. Establish a socket connection to the server
6. Send a request message and wait for the reply
7. Destroy the communicator, which closes the socket connection

Of primary concern are the activities that involve system calls, such as opening and reading files, creating and using sockets, and so on. The overhead incurred by these calls may not matter if the script is only executed infrequently, but for an application with high request rates it is necessary to minimize this overhead:

- A pre-configured property set eliminates the need to parse a property file in each request.
- Timeouts prevent a script from blocking indefinitely in case Ice encounters delays while performing socket operations.
- Registering a communicator avoids the need to create and destroy a communicator in every request.
- Be aware of the number of "round trips" (request-reply pairs) your script makes. For example, the script above uses `checkedCast` to verify that the remote Ice object supports the desired Slice interface. However, calling `checkedCast` causes the Ice run time to send a request to the server and await its reply, therefore this script is actually making two remote invocations. It is unnecessary to perform a checked cast if it is safe for the client to assume that the Ice object supports the correct interface, in which case using an `uncheckedCast` instead avoids the extra round trip.

Managing Property Sets in PHP

A PHP application can manually construct a property set for configuring its communicator. The Ice extension also provides a PHP-specific property set API that helps to minimize the overhead associated with initializing a communicator, allowing you to configure a default property set along with an unlimited number of named property sets (or *profiles*). You can populate a property set using a configuration file, command-line options, or both. Property sets are initialized using the normal Ice semantics: command-line options override any settings from a configuration file.

The Ice extension creates these property sets during web server startup, which means any subsequent changes you might make to the configuration have no effect until the web server is restarted. Also keep in mind that specifying a relative path name for a configuration file usually means the path name is evaluated relative to the web server's working directory.

Default Property Set in PHP

The INI directives `ice.config` and `ice.options` specify the configuration file and the command-line options for the default property set, respectively. These directives must appear in PHP's configuration file, which is usually named `php.ini`:

```
; Snippet from php.ini on Linux
extension=IcePHP.so
ice.config=/opt/MyApp/default.cfg
ice.options="--Ice.Override.Timeout=2000"
```

Profiles in PHP

Profiles are useful when several unrelated applications execute in the same web server, or when a script needs to choose among multiple configurations. To configure your profiles, add an `ice.profiles` directive to PHP's configuration file. The value of this directive is a file containing profile definitions:

```
; Snippet from php.ini on Linux
ice.profiles=/opt/MyApp/profiles
```

The profile definition file uses INI syntax:

```
[Production]
config=/opt/MyApp/prod.cfg
options="..."

[Debug]
config=/opt/MyApp/debug.cfg
options="--Ice.Trace.Network=3 ..."
```

The name of each profile is enclosed in square brackets. The configuration file and command-line options for each profile are defined using the `config` and `options` entries, respectively.

Using Property Sets in PHP

The `Ice\getProperties` function allows a script to obtain a copy of a property set. When called without an argument, or with an empty string, the function returns the default property set. Otherwise, the function expects the name of a configured profile and returns the property set associated with that profile. The return value is an instance of `Ice\Properties`, or `null` if no matching profile was found.

Note that the Ice extension always creates the default property set, which is empty if the `ice.config` and `ice.options` directives are not defined. Also note that changes a script might make to a property set returned by this function have no effect on other requests because the script is modifying a *copy* of the original property set.

Now we can modify our script to use `Ice\getProperties` and avoid the need to load a configuration file in each request:

PHP

```
<?php
require_once 'Ice.php';
require_once 'Hello.php';

try
{
    $data = new Ice\InitializationData;
    $data->properties = Ice\getProperties();
    $communicator = Ice\initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo\HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice\LocalException $ex)
{
    // Deal with exception...
}
?>
```

Security Considerations for Property Sets in PHP

Ice configuration properties may contain sensitive information such as the path name of the private key for an X.509 certificate. If multiple untrusted PHP applications run in the same web server, avoid the use of the default property set and choose sufficiently unique names for your named profiles. The Ice extension does not provide a means for enumerating the names of the configured profiles, therefore a malicious script would have to guess the name of a profile in order to examine its configuration properties.

To prevent a script from using the value of `ice.profiles` to open the profile definition file directly, enable the `ice.hide_profiles` direct ive to cause the Ice extension to replace the `ice.profiles` setting after it has processed the file. The `ice.hide_profiles` directive is enabled by default.

Timeouts in PHP

All twoway remote invocations made by a PHP script have synchronous semantics: the script does not regain control until Ice receives a reply from the server. As a result, we recommend configuring a suitable timeout value for all of your proxies as a defensive measure against network delays.

Registered Communicators in PHP

You can register a communicator to prevent it from being destroyed at the completion of a script. For example, a session-based PHP application can create a communicator for each new session and register it for reuse in subsequent requests of the same session. Reusing a communicator in this way avoids the overhead associated with creating and destroying a communicator in each request. Furthermore, it allows socket connections established by the Ice run time to remain open and available for use in another request.

Limitations of Registered Communicators in PHP

A communicator object is local to the process that created it, which in the case of PHP is usually a web server process. The usefulness of a registered communicator is therefore limited to situations in which an application can ensure that subsequent page requests are handled by the same web server process as the one that originally created the registered communicator. For example, registered communicators would not be appropriate in a typical CGI configuration because the CGI process terminates at the end of each request. A simple (but often impractical) solution is to configure your web server to use a single persistent process. The topic of configuring a web server to take advantage of registered communicators is outside the scope of this manual.

Using Registered Communicators in PHP

The API for registered communicators consists of three functions:

- `Ice\register($communicator, $name, $expires=0)`
  Registers a communicator with the given name. On success, the function returns true. If another communicator is already registered with the same name, the function returns false. The `expires` argument specifies a timeout value in minutes; if `expires` is greater than zero, the Ice extension automatically destroys the communicator if it has not been retrieved (via `Ice\find`) for the specified number of minutes. The default value (zero) means the communicator never expires, in which case the Ice extension only destroys the communicator when the current process terminates.
  It is legal to register a communicator with more than one name. In that case, the most recent value of expires takes precedence.
- `Ice\unregister($name)`
  Removes the registration for a communicator with the given name. Returns true if a match was found or false otherwise. Calling `Ice\unregister` does not cause the communicator to be destroyed; rather, the communicator is destroyed as soon as all pending requests that are currently using the communicator have completed. Destroying a registered communicator explicitly also removes its registration.
- `Ice\find($name)`
  Retrieves the communicator associated with the given name. Returns `null` if no match is found.

An application typically uses registered communicators as follows:

PHP

```
<?php
require_once 'Ice.php';

$communicator = Ice\find('MyCommunicator');
$expires = ...;
if($communicator == null)
{
    $communicator = Ice\initialize(...);
    Ice\register($communicator, 'MyCommunicator', $expires);
}

...
?>
```

Note that communicators consume resources such as threads, sockets, and memory, therefore an application should be designed to minimize the number of communicators it registers. Using a suitable expiration timeout prevents registered communicators from accumulating indefinitely.

A simple application that demonstrates the use of registered communicators can be found in the `Glacier2/hello` subdirectory of the PHP sample programs.

Security Considerations for Registered Communicators in PHP

There are risks associated with allowing untrusted applications to gain access to a registered communicator. For example, if a malicious script obtains a registered communicator that is configured with SSL credentials, the script could potentially make secure invocations as if it were the trusted script.

Registering a communicator with a sufficiently unique name reduces the chance that a malicious script could guess the communicator's name. For applications that make use of PHP's session facility, the session ID is a reasonable choice for a communicator name. The sample application in `Glacier2/hello` demonstrates this solution.

Lifetime of Object Factories in PHP

PHP reclaims all memory at the end of each request, which means any object factories that a script might have installed in a registered communicator are destroyed when the request completes even if the communicator is not destroyed. As a result, a script must install its object factories in a registered communicator for *every* request, as shown in the example below:

PHP

```
<?php
require 'Ice.php';

$communicator = Ice\find('MyCommunicator');
$expires = ...;
if($communicator == null)
{
    $communicator = Ice_initialize(...);
    Ice\register($communicator, 'MyCommunicator', $expires);
}

$communicator->addObjectFactory(new MyFactory, MyClass::ice_staticId());
...
?>
```

The Ice extension invokes the `destroy` method of each factory prior to the completion of a request.

See Also

- Properties and Configuration
- Connection Timeouts

Using Slice Checksums in PHP

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2php`, the `--checksum` option causes the compiler to generate checksums. The checksums are installed automatically when the PHP code is first parsed; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the two array objects using a comparison operator. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

PHP

```
$clientChecksums = Ice\sliceChecksums();
$serverChecksums = ...
foreach($clientChecksums as $key => $value)
{
    if(!isset($serverChecksums[$key]))
    {
        // No match found for type id!
    }
    else if($clientChecksums[$key] != $serverChecksums[$key])
    {
        // Checksum mismatch!
    }
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums

Example of a File System Client in PHP

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The PHP code shown here hardly differs from the code you would write for an ordinary PHP program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local PHP object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs.

We now have seen enough of the client-side PHP mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

PHP

```
<?php
require_once 'Ice.php';
require_once 'Filesystem.php';

// Recursively print the contents of directory "dir"
```

```
// in tree fashion. For files, show the contents of
// each file. The "depth" parameter is the current
// nesting level (for indentation).

function listRecursive($dir, $depth = 0)
{
    $indent = str_repeat("\t", ++$depth);

    $contents = $dir->_list(); // list is a reserved word in PHP

    foreach($contents as $i)
    {
        $dir = Filesystem\DirectoryPrxHelper::checkedCast($i);
        $file = Filesystem\FilePrxHelper::uncheckedCast($i);
        echo $indent . $i->name() .
($dir ? " (directory):" : " (file):") . "\n";
        if($dir)
        {
            listRecursive($dir, $depth);
        }
        else
        {
            $text = $file->read();
            foreach($text as $j)
            {
                echo $indent . "\t" . $j . "\n";
            }
        }
    }
}

$ic = null;
try
{
    // Create a communicator
    //
    $ic = Ice\initialize();

    // Create a proxy for the root directory
    //
    $obj = $ic->stringToProxy("RootDir:default -p 10000");

    // Down-cast the proxy to a Directory proxy
    //
    $rootDir = Filesystem\DirectoryPrxHelper::checkedCast($obj);

    // Recursively list the contents of the root directory
    //
    echo "Contents of root directory:\n";
    listRecursive($rootDir);
```

```
}
catch(Ice\LocalException $ex)
{
    print_r($ex);
}

if($ic)
{
    $ic->destroy(); // Clean up
```

```
}
?>
```

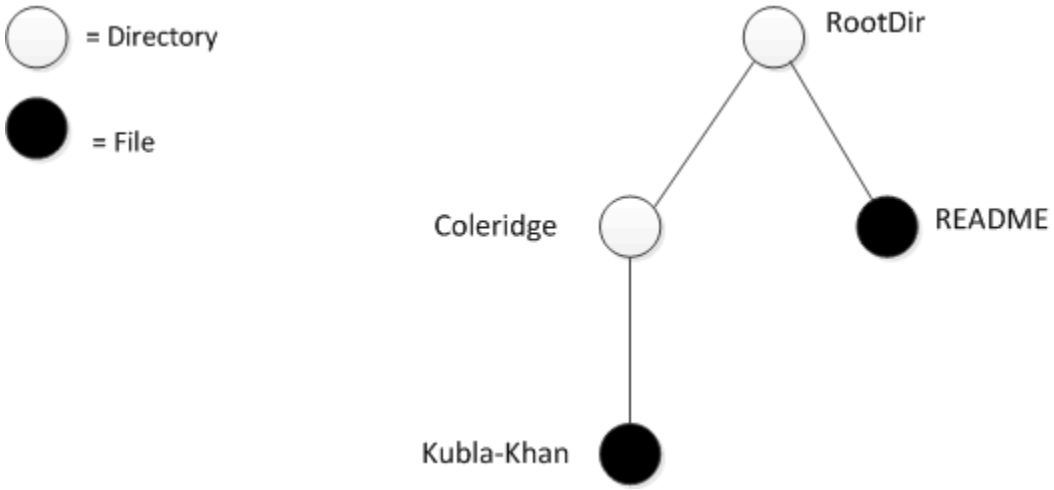
The program first defines the `listRecursive` function, which is a helper function to print the contents of the file system, and the main program follows. Let us look at the main program first:

1. The client first creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to the `Directory` interface and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code uses `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, and uses `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the proxy returned by `checkedCast`; if `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, `uncheckedCast` is sufficient to get a `File` proxy.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use `uncheckedCast` instead of `checkedCast` because `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of a two files and a a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client is not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Object Life Cycle
- IceGrid

Slice-to-PHP Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to PHP is identical to the mapping for these constructs without the `local` qualifier. The generated PHP code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

Data members on local Slice types (classes, exceptions and structs) are mapped to PHP just like the data members of the corresponding non-local Slice construct.

The rest of this section describes the mapping of the remaining local types to PHP:

PHP Mapping for Local Interfaces

On this page:

Mapped PHP Class

A Slice local interface is mapped to a PHP interface with the same name. For example:

Slice

```
module M
{
    local interface Example
    {
        ...
    }
}
```

is mapped to the PHP class `Example`:

PHP

```
namespace M
{
    interface Example
    {
        ...
    }
}
```

LocalObject in PHP

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to the native `stdClass` class in PHP.

Mapping for Local Interface Inheritance in PHP

Inheritance of local Slice interfaces is mapped to interface inheritance in PHP. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

PHP

```
namespace M
{
    interface A
    {
        ...
    }
    interface B extends A
    {
        ...
    }
    interface C extends A
    {
        ...
    }
    interface D extends B, C
    {
        ...
    }
}
```

PHP Mapping for Local Classes

On this page:

Mapped PHP Class

A local Slice class is mapped to a PHP class with the same name. For example:

Slice

```
module M
{
    local class Example
    {
        ...
    }
}
```

is mapped to the PHP class `Example`:

PHP

```
namespace M
{
    class Example
    {
        ...
    }
}
```

LocalObject in PHP

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to the native `stdClass` class in PHP.

Mapping for Local Interface Inheritance in PHP

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` and `implements` map to the same constructs in PHP. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

PHP

```
namespace M
{
    interface A
    {
        ...
    }
    interface B
    {
        ...
    }
    class C implements A, B
    {
        ...
    }
    class D extends C
    {
        ...
    }
}
```

PHP Mapping for Local Exceptions

On this page:

Mapped PHP Class

A local Slice exception is mapped to a PHP class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the PHP class `InitializationException`:

PHP

```
namespace Ice
{
    class InitializationException extends LocalException
    {
        ...
    }
}
```

Base Class for Local Exceptions in PHP

All mapped PHP classes for local exceptions extend the class `\Ice\LocalException`:

PHP

```
namespace Ice
{
    class LocalException extends Exception
    {
        ...
    }
}
```

`LocalException` derives from `\Ice\Exception`, which derives from PHP's native `Exception` class.

Mapping for Local Exception Inheritance in PHP

A local Slice exception can extend another Slice exception, which is mapped to class inheritance in PHP. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

PHP

```
namespace M
{
    class ErrorBase extends \Ice\LocalException
    {
        ...
    }
    class ResourceError extends ErrorBase
    {
        ...
    }
}
```

PHP Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to a PHP method with the same name. The mapping of operation parameters to PHP is identical to the client-side mapping for these parameters.

Unlike the client-side mapping, there is no mapped method with a trailing Context parameter.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

PHP

```
namespace M
{
    interface L
    {
        public function op($n, $s, $any, $m, $t, $newLSeq);
    }
}
```

Python Mapping

Topics

Initialization in Python

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `Ice.Communicator` object.

A `Communicator` is a local Python object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application.

You initialize the Ice run time by calling `Ice.initialize`, for example:

Python

```
import sys, Ice

communicator = Ice.initialize(sys.argv)
```

`Ice.initialize` accepts the argument list that is passed to the program by the operating system. The function scans the argument list for any command-line options that are relevant to the Ice run time; any such options are removed from the argument list so, when `Ice.initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your program, you must call `Communicator.destroy`. The `destroy` method is responsible for finalizing the Ice run time. In particular, in an Ice server, `destroy` waits for any operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your program to terminate without calling `destroy` first.

The general shape of our application becomes:

Python

```
import sys, traceback, Ice

status = 0
communicator = None
try:
    # correct but suboptimal, see below
    communicator = Ice.initialize(sys.argv)
    # ...
except:
    traceback.print_exc()
    status = 1

if communicator:
    # correct but suboptimal, see below
    communicator.destroy()

sys.exit(status)
```

This code is a little bit clunky, as we need to make sure the communicator gets destroyed in all paths, including when an exception is thrown.

Fortunately, `Communicator` implements the Python context manager protocol: this allows us to call `initialize` in a `with` statement, which destroys the communicator automatically, without an explicit call to the `destroy` method.

The preferred way to initialize the Ice run time in Python is therefore:

Python

```
import sys, Ice

with Ice.initialize(sys.argv) as communicator:
    # ...

# communicator is destroyed automatically at the end of the 'with'
statement
```

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-Python Mapping

The client-side Slice-to-Python mapping defines how Slice data types are translated to Python types, and how clients invoke operations, pass parameters, and handle errors. Much of the Python mapping is intuitive. For example, Slice sequences map to Python lists, so there is essentially nothing new you have to learn in order to use Slice sequences in Python.

The Python API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the Python mapping, you should need no more than the Slice definition of your application and knowledge of the Python mapping rules. In particular, looking through the generated code in order to discern how to use the Python mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **The `Ice` Module**
>
> All of the APIs for the Ice run time are nested in the `Ice` module, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` module are generated from Slice definitions; other parts of the `Ice` module provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` m odule throughout the remainder of the manual.
>
> A Python application can load the Ice run time using the `import` statement:
>
> ```
> import Ice
> ```
>
> If the statement executes without error, the Ice run time is loaded and available for use. You can determine the version of the Ice run time you have just loaded by calling the `stringVersion` function:
>
> ```
> icever = Ice.stringVersion()
> ```

Topics

Python Mapping for Identifiers

A Slice identifier maps to an identical Python identifier. For example, the Slice identifier `Clock` becomes the Python identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a Python keyword or is an identifier reserved by the Ice run time (such as `checkedCast`), the corresponding Python identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.

> You should try to avoid such identifiers as much as possible.

The mapping does not modify a Slice identifier that matches the name of a Python built-in function because it can always be accessed by its fully-qualified name. For example, the built-in function `hash` can also be accessed as `__builtin__.hash`.

See Also

- Lexical Rules
- Python Mapping for Modules
- Python Mapping for Built-In Types
- Python Mapping for Enumerations
- Python Mapping for Structures
- Python Mapping for Sequences
- Python Mapping for Dictionaries
- Python Mapping for Constants
- Python Mapping for Exceptions

Python Mapping for Modules

A Slice module maps to a Python module with the same name. The mapping preserves the nesting of the Slice definitions. Note that you can optionally use packages to gain further control over the generated code.

See Also

- Modules
- Python Mapping for Identifiers
- Python Mapping for Built-In Types
- Python Mapping for Enumerations
- Python Mapping for Structures
- Python Mapping for Sequences
- Python Mapping for Dictionaries
- Python Mapping for Constants
- Python Mapping for Exceptions
- Code Generation in Python

Python Mapping for Built-In Types

On this page:

Mapping of Slice Built-In Types to Python Types

The Slice built-in types are mapped to Python types as shown in this table:

Slice	Python
`bool`	`bool`
`short`	`int`
`int`	`int`
`long`	`long`
`float`	`double`
`double`	`double`
`string`	`string`

Although Python supports arbitrary precision in its integer types, the Ice run time validates integer values to ensure they have valid ranges for their declared Slice types.

String Mapping in Python 2

String values returned as the result of a Slice operation (including return values, out parameters, and data members) are always represented as instances of Python's 8-bit `string` type. These string values contain UTF-8 encoded strings unless the program has installed a string converter, in which case string values use the converter's native encoding instead.

Legal string input values for a remote Slice operation are shown below:

- `None`
  Ice marshals an empty string whenever `None` is encountered.
- 8-bit string objects
  Ice assumes that all 8-bit string objects contain valid UTF-8 encoded strings unless the program has installed a string converter, in which case Ice assumes that 8-bit string objects use the native encoding expected by the converter.
- Unicode objects
  Ice converts a Unicode object into UTF-8 and marshals it directly. If a string converter is installed, it is not invoked for Unicode objects.

String Mapping in Python 3

String values returned as the result of a Slice operation (including return values, out parameters, and data members) are always represented as instances of Python's Unicode-based `str` type. These string values contain UTF-8 encoded strings. The string converter facility is not used in Python 3.

Legal string input values for a remote Slice operation are shown below:

- `None`
  Ice marshals an empty string whenever `None` is encountered.
- String objects
  Ice converts strings to UTF-8 (if necessary) prior to marshaling.

See Also

Python Mapping for Enumerations

Python does not have an enumerated type, so a Slice enumeration is emulated using a Python class: the name of the Slice enumeration becomes the name of the Python class; for each enumerator, the class contains an attribute with the same name as the enumerator. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The generated Python class looks as follows:

Python

```
class Fruit(Ice.EnumBase):
    def valueOf(self, n):
        # ...
    valueOf = classmethod(valueOf)

    # ...

Fruit.Apple = ...
Fruit.Pear = ...
Fruit.Orange = ...
```

Each instance of the class has a `value` attribute providing the Slice value of the enumerator, and a `name` attribute that returns its name. The `valueOf` class method translates a Slice value into its corresponding enumerator, or returns `None` if no match is found.

Given the above definitions, we can use enumerated values as follows:

Python

```
f1 = Fruit.Apple
f2 = Fruit.Orange

if f1 == Fruit.Apple:               # Compare with constant
    # ...

if f1 == f2:                        # Compare two enums
    # ...

if f2.value == Fruit.Apple.value:   # Use Slice values
    # ...
elif f2.value == Fruit.Pear.value:
    # ...
elif f2.value == Fruit.Orange.value:
    # ...

Fruit.valueOf(1)  # Pear
```

As you can see, the generated class enables natural use of enumerated values. The `Fruit` class attributes are preinitialized enumerators that you can use for initialization and comparison.

Note that the generated class also defines a number of Python special methods, such as `__str__` and rich comparison operators, which we have not shown. The rich comparison operators compare the Slice value of the enumerator, which is not necessarily the same as its ordinal value.

Suppose we modify the Slice definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

We can use `valueOf` to examine the Slice values of the enumerators:

Python

```
Fruit.valueOf(0)  # Apple
Fruit.valueOf(1)  # None
Fruit.valueOf(3)  # Pear
Fruit.valueOf(4)  # Orange
```

See Also

- Enumerations
- Python Mapping for Identifiers
- Python Mapping for Modules
- Python Mapping for Built-In Types
- Python Mapping for Structures
- Python Mapping for Sequences
- Python Mapping for Dictionaries
- Python Mapping for Constants
- Python Mapping for Exceptions

Python Mapping for Structures

A Slice structure maps to a Python class with the same name. For each Slice data member, the Python class contains a corresponding attribute. For example, here is our `Employee` structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The Python mapping generates the following definition for this structure:

Python

```
class Employee(object):
    def __init__(self, number=0, firstName='', lastName=''):
        self.number = number
        self.firstName = firstName
        self.lastName = lastName

    def __eq__(self, other):
        # ...

    def __ne__(self, other):
        # ...

    def __str__(self):
        # ...

    def __hash__(self):
        # ...

    # ...
```

The constructor initializes each of the attributes to a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	None

`dictionary`	None
`class/interface`	None

You can also declare different default values for members of primitive and enumerated types.

The `__eq__` method returns true if all members of two structures are (recursively) equal, and `__ne__` returns true if any member differs.

The `__str__` method returns a string representation of the structure.

For structures that are also legal dictionary key types, the mapping also generates relational operators (`__lt__`, `__le__`, `__gt__`, `__ge__`) and a `__hash__` method. The `__hash__` method returns a hash value for the structure based on the value of all its data members.

See Also

- Structures
- Dictionaries
- Python Mapping for Identifiers
- Python Mapping for Modules
- Python Mapping for Built-In Types
- Python Mapping for Enumerations
- Python Mapping for Sequences
- Python Mapping for Dictionaries
- Python Mapping for Constants
- Python Mapping for Exceptions

Python Mapping for Sequences

On this page:

Default Sequence Mapping in Python

A Slice sequence maps by default to a Python list; the only exception is a sequence of bytes, which maps by default to a string in order to lower memory utilization and improve throughput. This use of native types means that the Python mapping does not generate a separate named type for a Slice sequence. It also means that you can take advantage of all the inherent functionality offered by Python's native types. For example, here is the definition of our `FruitPlatter` sequence once more:

Python

```
sequence<Fruit> FruitPlatter;
```

We can use the `FruitPlatter` sequence as shown below:

Python

```
platter = [ Fruit.Apple, Fruit.Pear ]
assert(len(platter) == 2)
platter.append(Fruit.Orange)
```

The Ice run time validates the elements of a tuple or list to ensure that they are compatible with the declared type; a `ValueError` exception is raised if an incompatible type is encountered.

Allowable Sequence Values in Python

Although each sequence type has a default mapping, the Ice run time allows a sender to use other types as well. Specifically, a tuple is also accepted for a sequence type that maps to a list, and in the case of a byte sequence, the sender is allowed to supply a tuple or list of integers as an alternative to a string.

> Using a string for a byte sequence bypasses the validation step and avoids an extra copy, resulting in much greater throughput than a tuple or list. For larger byte sequences, the use of a string is strongly recommended.

Furthermore, the Ice run time accepts objects that implement Python's buffer protocol as legal values for sequences of all primitive types except strings. For example, you can use the `array` module to create a buffer that is transferred much more efficiently than a tuple or list. Consider the two sequence values in the sample code below:

Python

```
import array
...
seq1 = array.array("i", [1, 2, 3, 4, 5])
seq2 = [1, 2, 3, 4, 5]
```

The values have the same on-the-wire representation, but they differ greatly in marshaling overhead because the buffer can be traversed more quickly and requires no validation.

Note that the Ice run time has no way of knowing what type of elements a buffer contains, therefore it is the application's responsibility to ensure that a buffer is compatible with the declared sequence type.

Customizing the Sequence Mapping in Python

The previous section described the allowable types that an application may use when sending a sequence. That kind of flexibility is not possible when receiving a sequence, because in this case it is the Ice run time's responsibility to create the container that holds the sequence.

As stated earlier, the default mapping for most sequence types is a list, and for byte sequences the default mapping is a string. Unless otherwise indicated, an application always receives sequences as the container type specified by the default mapping. If it would be more convenient to receive a sequence as a different type, you can customize the mapping by annotating your Slice definitions with metadata. The following table describes the metadata directives supported by the Python mapping:

Directive	Description
`python:seq:default`	Use the default mapping.
`python:seq:list`	Map to a Python list.
`python:seq:tuple`	Map to a Python tuple.

A metadata directive may be specified when defining a sequence, or when a sequence is used as a parameter, return value or data member. If specified at the point of definition, the directive affects all occurrences of that sequence type unless overridden by another directive at a point of use. The following Slice definitions illustrate these points:

Slice

```
sequence<int> IntList; // Uses list by default
["python:seq:tuple"] sequence<int> IntTuple; // Defaults to tuple

sequence<byte> ByteString; // Uses string by default
["python:seq:list"] sequence<byte> ByteList; // Defaults to list

struct S
{
    IntList i1; // list
    IntTuple i2; // tuple
    ["python:seq:tuple"] IntList i3; // tuple
    ["python:seq:list"] IntTuple i4; // list
    ["python:seq:default"] IntTuple i5; // list

    ByteString b1; // string
    ByteList b2; // list
    ["python:seq:list"] ByteString b3; // list
    ["python:seq:tuple"] ByteString b4; // tuple
    ["python:seq:default"] ByteList b5; // string
}

interface I
{
    IntList op1(ByteString s1, out ByteList s2);

    ["python:seq:tuple"]
    IntList op2(["python:seq:list"] ByteString s1,
                ["python:seq:tuple"] out ByteList s2);
}
```

The operation `op2` and the data members of structure `S` demonstrate how to override the mapping for a sequence at the point of use.

It is important to remember that these metadata directives only affect the receiver of the sequence. For example, the data members of structure `S` are populated with the specified sequence types only when the Ice run time unmarshals an instance of `S`. In the case of an operation, custom metadata affects the client when specified for the operation's return type and output parameters, whereas metadata affects the server for input parameters.

See Also

- Sequences
- Python Mapping for Identifiers
- Python Mapping for Modules
- Python Mapping for Built-In Types
- Python Mapping for Enumerations
- Python Mapping for Structures
- Python Mapping for Dictionaries
- Python Mapping for Constants
- Python Mapping for Exceptions

Python Mapping for Dictionaries

Here is the definition of our `EmployeeMap` once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Python mapping does not create a separate named type for this definition. Instead, *all* dictionaries are simply instances of Python's dictionary type. For example:

Python

```
em = {}

e = Employee()
e.number = 31
e.firstName = "James"
e.lastName = "Gosling"

em[e.number] = e
```

The Ice run time validates the elements of a dictionary to ensure that they are compatible with the declared type; a `ValueError` exception is raised if an incompatible type is encountered.

See Also

- Dictionaries
- Python Mapping for Identifiers
- Python Mapping for Modules
- Python Mapping for Built-In Types
- Python Mapping for Enumerations
- Python Mapping for Structures
- Python Mapping for Sequences
- Python Mapping for Constants
- Python Mapping for Exceptions

Python Mapping for Constants

Here are the constant definitions once more:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

The generated definitions for these constants are shown below:

Python

```
AppendByDefault = True
LowerNibble = 15
Advice = "Don't Panic!"
TheAnswer = 42
PI = 3.1416
FavoriteFruit = Fruit.Pear
```

As you can see, each Slice constant is mapped to a Python attribute with the same name as the constant.

Slice string literals that contain non-ASCII characters or universal character names are mapped to Python string literals with these characters replaced by their UTF-8 encoding as octal escapes. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

Python

```
Egg = "\305\223uf"
Heart = "c\305\223ur"
Banana = "\360\237\215\214"
```

See Also

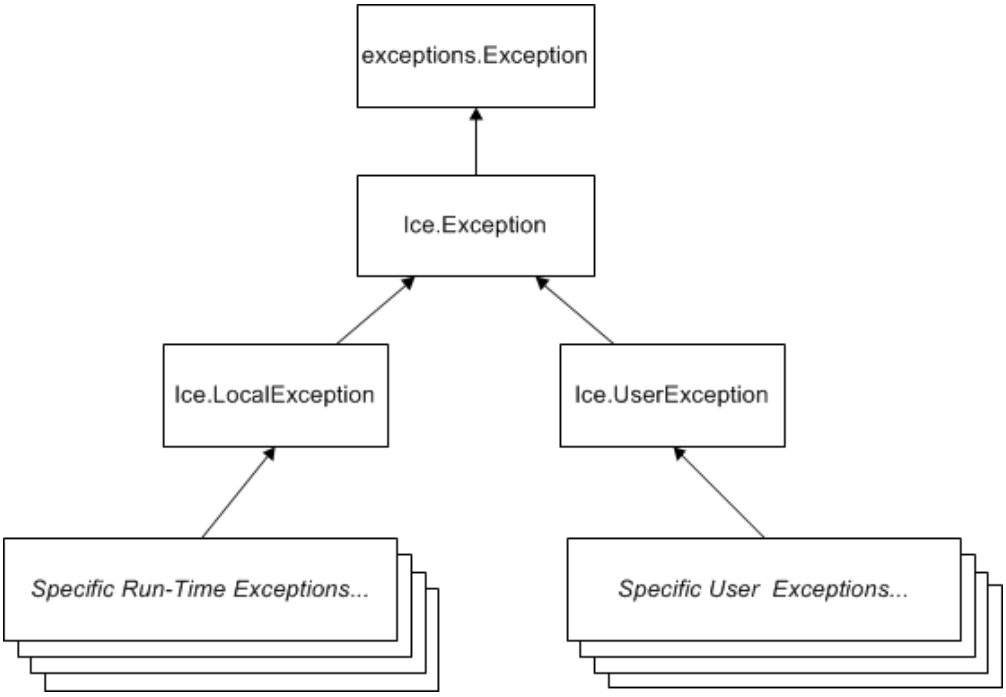
- Constants and Literals
- Python Mapping for Identifiers
- Python Mapping for Modules
- Python Mapping for Built-In Types

Python Mapping for Exceptions

On this page:

Inheritance Hierarchy for Exceptions in Python

The mapping for exceptions is based on the inheritance hierarchy shown below:



Inheritance structure for Ice exceptions.

The ancestor of all exceptions is `exceptions.Exception`, from which `Ice.Exception` is derived. `Ice.LocalException` and `Ice.UserException` are derived from `Ice.Exception` and form the base for all run-time and user exceptions.

Python Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map as follows:

Python

```
class GenericError(Ice.UserException):
    def __init__(self, reason=''):
        self.reason = reason

    def ice_id(self):
        # ...

    def __str__(self):
        # ...

class BadTimeVal(GenericError):
    def __init__(self, reason=''):
        GenericError.__init__(self, reason)

    def ice_id(self):
        # ...

    def __str__(self):
        # ...

class BadZoneName(GenericError):
    def __init__(self, reason=''):
        GenericError.__init__(self, reason)

    def ice_id(self):
        # ...

    def __str__(self):
        # ...
```

Each Slice exception is mapped to a Python class with the same name. The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception member corresponds to an attribute of the instance, which the constructor initializes to a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	None
`dictionary`	None

class/interface	None

You can also declare different default values for members of primitive and enumerated types. For derived exceptions, the constructor has one parameter for each of the base exception's data members, plus one parameter for each of the derived exception's data members, in base-to-derived order. As an example, although `BadTimeVal` and `BadZoneName` do not declare data members, their constructors still accept a value for the inherited data member `reason` in order to pass it to the constructor of the base exception `GenericError`.

Each exception also defines the `ice_id` method that returns the Slice type ID of the exception, and the special method `__str__` to return a stringified representation of the exception and its members.

All user exceptions are derived from the base class `Ice.UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice.UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice.LocalException`, and you can catch all Ice exceptions with a handler for `Ice.Exception`.

Optional Data Members

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice.Unset` to indicate that the member is unset. A well-behaved program must test an optional data member before using its value:

Python

```
try:
    ...
except ex:
    if ex.optionalMember:
        print("optionalMember = " + str(ex.optionalMember))
    else:
        print("optionalMember is unset")
```

The `Ice.Unset` marker value has different semantics than `None`. Since `None` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to `None` is considered to be set. If you need to distinguish between an unset value and a value set to `None`, you can do so as follows:

Python

```
try:
    ...
except ex:
    if ex.optionalMember is Ice.Unset:
        print("optionalMember is unset")
    elif ex.optionalMember is None:
        print("optionalMember is None")
    else:
        print("optionalMember = " + str(ex.optionalMember))
```

Python Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice.LocalException` (which, in turn, derives from `Ice.Exception`).

By catching exceptions at the appropriate point in the inheritance hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice.LocalException`

This is the root of the inheritance tree for run-time exceptions.

- `Ice.UserException`
  This is the root of the inheritance tree for user exceptions.
- `Ice.TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `Ice.ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- Python Mapping for Identifiers
- Python Mapping for Modules
- Python Mapping for Built-In Types
- Python Mapping for Enumerations
- Python Mapping for Structures
- Python Mapping for Sequences
- Python Mapping for Dictionaries
- Python Mapping for Constants
- Optional Data Members
- Versioning
- Object Life Cycle

Python Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Classes in Python

On the client side, a Slice interface maps to a Python class with methods that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Python mapping generates the following definition for use by the client:

Python

```
class SimplePrx(Ice.ObjectPrx):
    def op(self, context=None):
        # ...

    def ice_staticId():
        return '...'
    ice_staticId = staticmethod(ice_staticId)

    # ...
```

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. In the preceding example, we find that the operation `op` has been mapped to the method `op`. Note that `op` accepts an optional trailing parameter `_ctx` representing the operation context. This parameter is a Python dictionary for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the context parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of `None` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Another method defined by every proxy class is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Interface Inheritance in Python

Inheritance relationships among Slice interfaces are maintained in the generated Python classes. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

The generated code for `CPrx` reflects the inheritance hierarchy:

Python

```
class CPrx(APrx, BPrx):
    ...
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

`Ice.ObjectPrx` Class in Python

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

Python

```
class ObjectPrx(object):
    def equals(self, other):
    def ice_getIdentity(self):
    def ice_isA(self, id):
    def ice_ids(self):
    def ice_id(self):
    def ice_ping(self):
    # ...
```

The methods behave as follows:

- **`equals`**
  This method compares two proxies for equality. Note that all aspects of proxies are compared by this method, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
}
```

To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

Python

```
proxy1 = ...
proxy2 = ...
id1 = proxy1.ice_getIdentity()
id2 = proxy2.ice_getIdentity()

if id1 == id2:
    # proxy1 and proxy2 denote the same object
else:
    # proxy1 and proxy2 denote different objects
```

- **ice_isA**
  The ice_isA method determines whether the object denoted by the proxy supports a specific interface. The argument to ice_isA is a type ID. For example, to see whether a proxy of type ObjectPrx denotes a Printer object, we can write:

  Python

  ```
  proxy = ...
  if proxy != None and proxy.ice_isA("::Printer"):
      # proxy denotes a Printer object
  else:
      # proxy denotes some other type of object
  ```

  Note that we are testing whether the proxy is None before attempting to invoke the ice_isA method. This avoids getting a run-time error if the proxy is None.

- **ice_ids**
  The ice_ids method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- **ice_id**
  The ice_id method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type BasePrx, with a static type ID of ::Base, the return value of ice_id might be ::Base, or it might something more derived, such as ::Derived.

- **ice_ping**
  The ice_ping method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

The `ice_isA`, `ice_ids`, `ice_id`, and `ice_ping` methods are remote operations and therefore support an optional trailing parameter representing a request context. Also note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Casting Proxies in Python

The Python mapping for a proxy also generates two static methods:

Python

```
class SimplePrx(Ice.ObjectPrx):
    # ...

    def checkedCast(proxy, facet=''):
        # ...
    checkedCast = staticmethod(checkedCast)

    def uncheckedCast(proxy, facet=''):
        # ...
    uncheckedCast = staticmethod(uncheckedCast)
```

The method names `checkedCast` and `uncheckedCast` are reserved for use in proxies. If a Slice interface defines an operation with either of those names, the mapping escapes the name in the generated proxy by prepending an underscore. For example, an interface that defines an operation named `checkedCast` is mapped to a proxy with a method named `_checkedCast`.

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is `None`), the cast returns `None`.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

Python

```
obj = ...       # Get a proxy from somewhere...

simple = SimplePrx.checkedCast(obj)
if simple != None:
    # Object supports the Simple interface...
else:
    # Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

Python

```
obj = ...                                # Get proxy...
process = ProcessPrx.uncheckedCast(obj) # No worries...
process.launch(40, 60)                   # Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

Using Proxy Methods in Python

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

Python

```
proxy = communicator.stringToProxy(...)
proxy = proxy.ice_invocationTimeout(10000)
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

Python

```
base = communicator.stringToProxy(...)
hello = Demo.HelloPrx.checkedCast(base)
hello = hello.ice_invocationTimeout(10000) # Type is preserved
hello.sayHello()
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in Python

Proxy objects support comparison using the built-in relational operators as well as the `cmp` function. Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison tests for *proxy* identity , *not* object identity. A common mistake is to write code along the following lines:

Python

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if p1 != p2:
    # p1 and p2 denote different objects       # WRONG!
else:
    # p1 and p2 denote the same object         # Correct
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each uses a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` module:

Python

```
def proxyIdentityCompare(lhs, rhs)
def proxyIdentityAndFacetCompare(lhs, rhs)
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

Python

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if Ice.proxyIdentityCompare(p1, p2) != 0:
    # p1 and p2 denote different objects       # Correct
else:
    # p1 and p2 denote the same object         # Correct
```

The function returns 0 if the identities are equal, `1` if `p1` is less than `p2`, and `1` if `p1` is greater than `p2`. (The comparison uses `name` as the major sort key and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Python Mapping for Operations
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

Python Mapping for Operations

On this page:

Basic Python Mapping for Operations

As we saw in the Python mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding method with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

Python

```
node = ...          # Initialize proxy
name = node.name()  # Get name via RPC
```

Normal and `idempotent` Operations in Python

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent  string op2();
}
```

The proxy class for this is:

Python

```
class ExamplePrx(Ice.ObjectPrx):
    def op1(self, context=None):
        # ...

    def op2(self, context=None):
        # ...
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to look the same.

Passing Parameters in Python

In-Parameters in Python

All parameters are passed by reference in the Python mapping; it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy for this definition:

Python

```
class ClientToServerPrx(Ice.ObjectPrx):
    def op1(self, i, f, b, s, context=None):
        # ...

    def op2(self, ns, ss, st, context=None):
        # ...

    def op3(self, proxy, context=None):
        # ...
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

Python

```
p = ...                                  # Get proxy...

p.op1(42, 3.14f, True, "Hello world!")  # Pass simple literals

i = 42
f = 3.14f
b = True
s = "Hello world!"
p.op1(i, f, b, s)                        # Pass simple variables

ns = NumberAndString()
ns.x = 42
ns.str = "The Answer"
ss = [ "Hello world!" ]
st = {}
st[0] = ns
p.op2(ns, ss, st)                        # Pass complex variables

p.op3(p)                                 # Pass proxy
```

Out-Parameters in Python

As in Java, Python functions do not support reference arguments. That is, it is not possible to pass an uninitialized variable to a Python function in order to have its value initialized by the function. The Java mapping overcomes this limitation with the use of *holder classes* that represent each `out` parameter. The Python mapping takes a different approach, one that is more natural for Python users.

The semantics of `out` parameters in the Python mapping depend on whether the operation returns one value or multiple values. An operation returns multiple values when it has declared multiple `out` parameters, or when it has declared a non-`void` return type and at least one `out` parameter.

If an operation returns multiple values, the client receives them in the form of a *result tuple*. A non-`void` return value, if any, is always the first element in the result tuple, followed by the `out` parameters in the order of declaration.

If an operation returns only one value, the client receives the value itself.

Here again are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    int op1(out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Python mapping generates the following code for this definition:

Python

```
class ServerToClientPrx(Ice.ObjectPrx):
    def op1(self, context=None):
        # ...

    def op2(self, context=None):
        # ...

    def op3(self, context=None):
        # ...
```

Given a proxy to a `ServerToClient` interface, the client code can receive the results as in the following example:

Python

```
p = ...              # Get proxy...
i, f, b, s = p.op1()
ns, ss, st = p.op2()
stcp = p.op3()
```

The operations have no `in` parameters, therefore no arguments are passed to the proxy methods. Since `op1` and `op2` return multiple values, their result tuples are unpacked into separate values, whereas the return value of `op3` requires no unpacking.

Parameter Type Mismatches in Python

Although the Python compiler cannot check the types of arguments passed to a function, the Ice run time does perform validation on the arguments to a proxy invocation and reports any type mismatches as a `ValueError` exception.

Null Parameters in Python

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `None`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `None` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, it makes no difference to the receiver whether you send a string as `None` or as an empty string: either way, the receiver sees an empty string.

Optional Parameters in Python

Optional parameters use the same mapping as required parameters. The only difference is that `Ice.Unset` can be passed as the value of an optional parameter or return value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

A client can invoke this operation as shown below:

Python

```
i, v = proxy.execute("--file log.txt")
i, v = proxy.execute(Ice.Unset)

if v:
    print("value = " + str(v)) # v is set to a value
```

A well-behaved program must always test an optional parameter prior to using its value. Keep in mind that the `Ice.Unset` marker value has different semantics than `None`. Since `None` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional parameter is set. An optional parameter set to `None` is considered to be set. If you need to distinguish between an unset parameter and a parameter set to `None`, you can do so as follows:

Python

```
if optionalParam is Ice.Unset:
    print("optionalParam is unset")
elif optionalParam is None:
    print("optionalParam is None")
else:
    print("optionalParam = " + str(optionalParam))
```

Exception Handling in Python

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as Python exceptions, so you can simply enclose one or more operation invocations in a `try-except` block:

Python

```
child = ...       # Get child proxy...

try:
    child.askToCleanUp()
except Tantrum, t:
    print "The child says:", t.reason
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will usually be handled by exception handlers higher in the hierarchy. For example:

Python

```
import traceback, Ice

try:
    child = ...        # Get child proxy...
    try:
        child.askToCleanUp()
        child.praise() # Give positive feedback...
    except Tantrum, t:
        print "The child says:", t.reason
        child.scold()  # Recover from error...
except Ice.LocalException:
    traceback.print_exc()
```

See Also

- Operations
- Hello World Application
- Slice for a Simple File System

Python Mapping for Classes

On this page:

Basic Python Mapping for Classes

A Slice class maps to a Python class with the same name. The generated class contains an attribute for each Slice data member (just as for structures and exceptions). Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

The Python mapping generates the following code for this definition:

Python

```
class TimeOfDay(Ice.Value):
    def __init__(self, hour=0, minute=0, second=0):
        # ...
        self.hour = hour
        self.minute = minute
        self.second = second

    def ice_id(self):
  return '::M::TimeOfDay'

    @staticmethod
    def ice_staticId():
        return '::M::TimeOfDay'

    # ...
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice.Value`. This means that all classes implicitly inherit from `Ice.Value`, which is the ultimate ancestor of all classes. Note that `Ice.Value` is *not* the same as `Ice.ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The constructor defines an attribute for each Slice data member.
3. The class defines the method `ice_id` and static method `ice_staticId`.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice.Value` in Python

Like interfaces, classes implicitly inherit from a common base class, `Ice.Value`. However classes inherit from `Ice.Value` instead of `Ice.ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`Ice.Value` contains a number of methods:

Python

```
class Value(object):
    def ice_id(self):
        # ...

 @staticmethod
    def ice_staticId():
        # ...

    def ice_preMarshal(self):
        # ...

    def ice_postUnmarshal(self):
        # ...

 def ice_getSlicedData(self):
        # ...
```

The member functions of `Ice.Value` behave as follows:

- `ice_id`
  This method returns the actual run-time type ID of the object. If you call `ice_id` through a reference to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.
- `ice_staticId`
  This method is generated in each class and returns the static type ID of the class.
- `ice_preMarshal`
  The Ice run time invokes this method prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this method after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `None` otherwise.

Note that neither `Ice.Value` nor the generated class override `__hash__` and `__eq__`, so the default implementations apply.

Class Data Members in Python

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding attribute.

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice.Unset` to indicate that the member is unset. A well-behaved program must test an optional data member before using its value:

Python

```
obj = ...
if obj.optionalMember:
    print("optionalMember = " + str(ex.optionalMember))
else:
    print("optionalMember is unset")
```

The `Ice.Unset` marker value has different semantics than `None`. Since `None` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to `None` is considered to be set. If you need to distinguish between an unset value and a value set to `None`, you can do so as follows:

Python

```
obj = ...
if obj.optionalMember is Ice.Unset:
    print("optionalMember is unset")
elif obj.optionalMember is None:
    print("optionalMember is None")
else:
    print("optionalMember = " + str(ex.optionalMember))
```

Although Python provides no standard mechanism for restricting access to an object's attributes, by convention an attribute whose name begins with an underscore signals the author's intent that the attribute should only be accessed by the class itself or by one of its subclasses. You can employ this convention in your Slice classes using the `protected` metadata directive. The presence of this directive causes the Slice compiler to prepend an underscore to the mapped name of the data member. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
}
```

The Slice compiler produces the following generated code for this definition:

Python

```
class TimeOfDay(Ice.Value):
    def __init__(self, hour=0, minute=0, second=0):
        # ...
        self._hour = hour
        self._minute = minute
        self._second = second

    # ...
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

Class Constructors in Python

Classes have a constructor that assigns to each data member a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	None
`dictionary`	None
`class/interface`	None

You can also declare different default values for data members of primitive and enumerated types.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

You can invoke this constructor in one of two ways:

- Provide values for all members, including optional members, in the order of declaration:

Python

```
t = TimeOfDay(12, 33, 45)
t2 = TimeOfDay(14, 7) # second defaults to 0
```

Pass `Ice.Unset` as the value of any optional member you want to be unset.

- Used named arguments to specify values for certain members and in any order:

Python

```
t = TimeOfDay(minute=33, hour=12)
```

Class Operations in Python

Deprecated Feature
Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

With the Python mapping, operations in classes are not mapped at all into the corresponding Python class. The generated Python class is the same whether the Slice class has operations or not.

The Slice to Python compiler also generates a separate `<class-name>Disp` class, which can be used to implement an Ice object with these operations. For example:

Slice

```
class FormattedTimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;
    string format();
}
```

results in the following generated code:

Python

```
class FormattedTimeOfDay(Ice.Value):
     # ... operation format() not mapped at all here

# Disp class for servant implementation
class FormattedTimeOfDayDisp(Ice.Object):
     # ...
```

The `Disp` class is the Python *skeleton class* for this Slice class. Skeleton classes are described in the Server-Side Python Mapping for

Interfaces.

Value Factories in Python

> While value factories were previously necessary when using classes with operations (a now deprecated feature), value factories may be used for any kind of class and are *not* deprecated.

Value factories allow you to create classes derived from the Python class generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
interface Time
{
    TimeOfDay get();
}
```

The Ice run time will by default create and return a plain `TimeOfDay` instance.

If you wish, you can create your own custom derived class, and tell Ice to create and return these instances instead. For example:

Python

```
class CustomTimeOfDay(TimeOfDay):
    def format(self):
       # prints formatted data members
       pass
```

You then create and register a value factory for your custom class with your Ice communicator:

Python

```
def ValueFactory(Ice.ValueFactory):
    if type == TimeOfDay.ice_staticId():
        return TimeOfDayI()
    assert(False)
    return None

communicator = ...
communicator.getValueFactoryManager().add(ValueFactory(),
TimeOfDay.ice_staticId()))
```

See Also

- Classes
- Type IDs
- Optional Data Members
- Python Mapping for Operations
- The Current Object

- Value Factories

Asynchronous Method Invocation (AMI) in Python

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

Topics

AMI in Python with Futures

On this page:

Basic Asynchronous API in Python

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in Python

In addition to the synchronous proxy method, the Python mapping generates the following asynchronous proxy method:

Python

```
def getNameAsync(self, number, context=None)
```

As you can see, the `getName` operation generates a `getNameAsync` method, which optionally accepts a per-invocation context. `getNameAsync` sends (or queues) an invocation of `getName`, and does not block the calling thread. It returns an instance of `Ice.InvocationFuture` that you can use in a number of ways, including blocking to obtain the result, configuring an action to be executed when the result becomes available, and canceling the invocation.

Here's an example that calls `getNameAsync`:

Python

```
e = EmployeePrx.checkedCast(...)
f = e.getNameAsync(99)

# Continue to do other things here...

name = f.result()
```

Because `getNameAsync` does not block, the calling thread can do other things while the operation is in progress.

An asynchronous proxy method uses the same parameter mapping as for synchronous operations; the only difference is that the result (if any) is returned via an `InvocationFuture`. For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The generated code looks like this:

Python

```
def opAsync(self, inp1, inp2, context=None)
```

Now let's call `add_done_callback` to demonstrate one way of asynchronously executing an action when the invocation completes:

Python

```
p.opAsync(42, "value for inp2").add_done_callback(lambda future: ret,
outp1, outp2 = future.result())
```

As with the synchronous mapping, an operation that returns multiple values supplies its result as a tuple. The completion callback, in this case a lambda function, receives the future as its argument and extracts the values from the result tuple.

Asynchronous Exception Semantics in Python

If an invocation raises an exception, the exception can be obtained from the future in several ways:

- Call `result` on the future; `result` raises the exception directly
- Call `exception` on the future; `exception` returns the exception object

The exception is provided by the future, even if the actual error condition for the exception was encountered during the call to the `opAsync` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that handles the future (instead of being present twice, once where the `opAsync` method is called, and again where the future is handled).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

`Future` Classes in Python

Ice provides two future classes: `Ice.Future` and `Ice.InvocationFuture`. Asynchronous proxy invocations return an instance of `InvocationFuture`, which is a subclass of `Future`. The API for `Ice.Future` is similar to that of Python's `asyncio.Future` and `concurrent.futures.Future` classes, while `InvocationFuture` adds some Ice-specific methods that clients may find useful.

Python

```
class Future(...):
    def cancel(self)
    def cancelled(self)
    def running(self)
    def done(self)

    def add_done_callback(self, fn)

    def result(self, timeout=None)
    def exception(self, timeout=None)

    def set_result(self, result)
    def set_exception(self, ex)

    def completed(result)
    completed = staticmethod(completed)

class InvocationFuture(Future):
    def add_done_callback_async(self, fn)

    def is_sent(self)
    def is_sent_synchronously(self)
    def add_sent_callback(self, fn)
    def add_sent_callback_async(self, fn)
    def sent(self, timeout=None)
    def set_sent(self, sentSynchronously)

    def communicator(self)
    def connection(self)
    def proxy(self)
    def operation(self)
```

The `Future` methods have the following semantics:

- `cancel(self)`
  This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. `cancel` is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning `done` returns true, and the result of the invocation is an `Ice.InvocationCanceledException`.
- `cancelled(self)`
  This method returns frue if the invocation was cancelled via a call to `cancel`, or false otherwise.
- `running(self)`
  This method returns true if the invocation has not yet completed or been cancelled, or false otherwise.
- `done(self)`
  This method returns true if the invocation has completed (either successfully or exceptionally) or has been cancelled, or false otherwise.
- `add_done_callback(self, fn)`
  This method registers a callback to be executed when the invocation completes, either successfully or exceptionally. The callback function receives the future as its only argument. If the invocation is already completed at the time `add_done_callback` is called,

the callback method is invoked recursively from the calling thread, otherwise the callback method is invoked in the thread that completes the invocation.

- `result(self, timeout=None)`
  This method returns the result of the invocation. If an optional timeout is provided, the method will block for up to the given timeout waiting for the invocation to complete and raises `Ice.TimeoutException` if the timeout expires without completion. If no timeout is provided, the method blocks indefinitely. If the invocation completes with an exception, the method raises the exception directly. For a Slice operation declared with a `void` return type, the method returns `None` upon successful completion.

- `exception(self, timeout=None)`
  This method returns the exception that completed the invocation, or `None` if the invocation completed successfully. If an optional timeout is provided, the method will block for up to the given timeout waiting for the invocation to complete and raises `Ice.TimeoutException` if the timeout expires without completion. If no timeout is provided, the method blocks indefinitely.

- `set_result(self, result)`
  This method completes the invocation successfully using the given result. Calling this method has no effect if the invocation is already completed.

- `set_exception(self, ex)`
  This method completes the invocation exceptionally using the given exception. Calling this method has no effect if the invocation is already completed.

- `completed(result)`
  This static convenience method returns an instance of `Ice.Future` that is already completed successfully with the given result.

The `InvocationFuture` methods have the following semantics:

- `add_done_callback_async(self, fn)`
  This method's semantics differ from that of `add_done_callback` in the situation where the future is already completed. When you call `add_done_callback_async` and the future is already completed, the callback will be invoked by an Ice thread (or by a dispatcher if one is configured).

- `is_sent(self)`
  When you call an asynchronous proxy method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `is_sent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `is_sent` returns false.

- `is_sent_synchronously(self)`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `is_sent_synchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

- `add_sent_callback(self, fn)`
  This method registers a callback to be executed when the invocation has been sent. The callback function receives two arguments: the future object and a boolean indicating whether the invocation was sent synchronously. If the invocation is already sent at the time `add_sent_callback` is called, the callback method is invoked recursively from the calling thread. Otherwise, the callback method is invoked by an Ice thread (or by a dispatcher if one is configured).

- `add_sent_callback_async(self, fn)`
  This method's semantics differ from that of `add_sent_callback` in the situation where the invocation is already sent. When you call `add_sent_callback_async` and the invocation is already sent, the callback will be invoked by an Ice thread (or by a dispatcher if one is configured).

- `sent(self, timeout=None)`
  This method waits for the invocation to be sent and returns a boolean indicating whether the invocation was sent synchronously. If an optional timeout is provided, the method will block for up to the given timeout waiting for the invocation to be sent and raises `Ice.TimeoutException` if the timeout expires beforehand. If no timeout is provided, the method blocks indefinitely. If the invocation completes with an exception, the method raises the exception directly.

- `set_sent(self, sentSynchronously)`
  This method marks the invocation as sent, and the boolean argument indicates whether it was sent synchronously.

- `communicator(self)`
  This method returns the communicator that sent the invocation.

- `connection(self)`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change

unexpectedly. The `getConnection` method only returns a non-nil value when the `AsyncResult` object is obtained by calling `begin_flushBatchRequests` on a `Connection` object.

- `proxy(self)`
  This method returns the proxy that was used to call the asynchronous proxy method, or `None` if the future was not obtained via an asynchronous proxy invocation.
- `operation(self)`
  This method returns the name of the operation.

Python 3.5 Features

Ice's future types provide some additional features when using Python 3.5 or later.

`asyncio` Integration

The `Ice.wrap_future` function wraps an Ice future object with an instance of `asyncio.Future`. The function accepts an `Ice.Future` object and returns an `asyncio.Future` object. Since `Ice.Future` objects support use in multi-threaded applications, `wrap_future` ensures that the resulting `asyncio.Future` object is completed in a thread-safe manner.

Awaitable Objects

`Ice.Future` is an *awaitable* object, meaning an instance can be used as the target of the `await` keyword. Note however that your chosen event loop implementation must also support `Ice.Future` objects. For example, attempting to call `await` on an `Ice.Future` while using the `asyncio` event loop will result in an error because `asyncio`'s event loop doesn't support "foreign" future types.

One situation where `Ice.Future` objects can be awaited is in a servant dispatch method that is implemented as a coroutine.

Polling for Completion in Python

The `InvocationFuture` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0
while not file.eof():
    bytes = file.read(chunkSize)  # Read a chunk
    ft.send(offset, bytes)        # Send the chunk
    offset += len(bytes.length)
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time

doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0

results = []
numRequests = 5

while not file.eof():
    bytes = file.read(chunkSize) # Read a chunk

    # Send up to numRequests + 1 chunks asynchronously.
    f = ft.sendAsync(offset, bytes)
    offset += len(bytes)

    # Wait until this request has been passed to the transport.
    f.sent()
    results.append(f)

    # Once there are more than numRequests, wait for the least
    # recent one to complete.
    while len(results) > numRequests:
        f = results[0]
        del results[0]
        f.result()

# Wait for any remaining requests to complete.
while len(results) > 0:
    f = results[0]
    del results[0]
    f.result()
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Asynchronous Oneway Invocations in Python

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous proxy method on a oneway proxy for an operation that returns values or raises a user exception, the method throws `TwowayOnlyException`.

The future returned for a oneway invocation completes as soon as the request is successfully written to the client-side transport. The future completes exceptionally if an error occurs before the request is successfully written.

Flow Control in Python

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `InvocationFuture.is_sent_synchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `InvocationFuture.is_sent_synchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

You can supply a sent callback to be notified when the request was successfully sent:

Python

```
def sentCallback(future, sentSynchronously):
   # The request was sent, send another!

proxy = ...

future = proxy.doSomethingAsync()
future.add_sent_callback(sentCallback)
```

The `add_sent_callback` method has the following semantics:

- If the Ice run time was able to pass the entire request to the local transport immediately, the action will be invoked from the current thread and the `sentSynchronously` argument will be true.
- If Ice wasn't able to write the entire request without blocking, the action will eventually be invoked from an Ice thread pool thread and the `sentSynchronously` argument will be false.

Asynchronous Batch Requests in Python

You can invoke operations via batch oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous proxy method on a oneway proxy for an operation that returns values or raises a user exception, the method throws `TwowayOnlyException`.

The future returned for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The future completes exceptionally if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides an asynchronous version of this method so you can flush batch requests asynchronously.

`ice_flushBatchRequestsAsync` is a proxy method that flushes any batch requests queued by that proxy, without blocking the calling thread.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `InvocationFuture.connection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency Semantics for AMI in Python

For the `InvocationFuture` returned by an asynchronous proxy method, the Ice run time invokes `set_result` or `set_exception` from

an Ice thread pool thread. When you register an action with `add_done_callback`, the thread in which your action executes depends on the completion status of the future. If the future is already complete at the time you call `add_done_callback`, the callback function will be invoked immediately in the calling thread. If the future is not yet complete when you call `add_done_callback`, the action will eventually execute in an Ice thread pool thread.

The semantics are slightly different when you register an action with `add_done_callback_async`: the action is always executed in an Ice thread pool thread regardless of the completion status of the future at the time of the call.

> If a dispatcher is configured, the Ice thread pool thread delegates the execution of the action to the dispatcher.

Refer to the flow control discussion for information about the concurrency semantics of the flow control methods.

See Also

- Python Mapping for Operations
- Request Contexts
- Batched Invocations

AMI in Python with AsyncResult

> The AMI mapping using the AsyncResult API is deprecated and provided only for backward compatibility. New applications should use the Futures API.

On this page:

Basic Asynchronous API in Python

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in Python

Besides the synchronous proxy methods, the Python mapping generates the following asynchronous proxy methods:

Python

```
def begin_getName(self, number, _response=None, _ex=None,
_sent=None, _ctx=None)
def end_getName(self, result)
```

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. The `begin_ method` optionally accepts a per-invocation context and callbacks.

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

Python

```
e = EmployeePrx.checkedCast(...)
r = e.begin_getName(99)

# Continue to do other things here...

name = e.end_getName(r)
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `AsyncResult`. This value contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResult` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. The `end_` method accepts the `AsyncResult` object as its only argument and returns the out-parameters using the same semantics as for regular synchronous invocations. For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

Python

```
def begin_op(self, inp1, inp2, ...)
def end_op(self, result)
```

The call to `end_op` returns the following tuple:

Python

```
doubleValue, outp1, outp2 = p.end_op(result)
```

Asynchronous Exception Semantics in Python

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `RuntimeError`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

AsyncResult Class in Python

The AsyncResult that is returned by the begin_ method encapsulates the state of the asynchronous invocation:

Python

```
class AsyncResult:
    def cancel()

    def getCommunicator()
    def getConnection()
    def getProxy()
    def getOperation()

    def isCompleted()
    def waitForCompleted()

    def isSent()
    def waitForSent()

    def throwLocalException()

    def sentSynchronously()
```

The methods have the following semantics:

- cancel()
  This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. cancel is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning isCompleted returns true, and the result of the invocation is an Ice.InvocationCanceledException.
- getCommunicator()
  This method returns the communicator that sent the invocation.
- getConnection()
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The getConnection method only returns a non-nil value when the AsyncResult object is obtained by calling begin_flushBatchRequests on a Connection object.
- getProxy()
  This method returns the proxy that was used to call the begin_ method, or nil if the AsyncResult object was not obtained via an asynchronous proxy invocation.
- getOperation()
  This method returns the name of the operation.
- isCompleted()
  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the end_ method will not block the caller. Otherwise, if the result is not yet available, the method returns false.
- waitForCompleted()
  This method blocks the caller until the result of an invocation becomes available.
- isSent()
  When you call the begin_ method, the Ice run time attempts to write the corresponding request to the client-side transport. If the

transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.

- `waitForSent()`
  This method blocks the calling thread until a request has been written to the client-side transport, or an exception occurs. After `waitForSent` returns, `isSent` returns true if the request was successfully written to the client-side transport, or false if an exception occurred. In the case of a failure, you can call the corresponding `end_` method or `throwLocalException` to obtain the exception.
- `throwLocalException()`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.
- `sentSynchronously()`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

Polling for Completion in Python

The `AsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0
while not file.eof():
    bytes = file.read(chunkSize)  # Read a chunk
    ft.send(offset, bytes)        # Send the chunk
    offset += len(bytes.length)
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0

results = []
numRequests = 5

while not file.eof():
    bytes = file.read(chunkSize) # Read a chunk

    # Send up to numRequests + 1 chunks asynchronously.
    r = ft.begin_send(offset, bytes)
    offset += len(bytes)

    # Wait until this request has been passed to the transport.
    r.waitForSent()
    results.append(r)

    # Once there are more than numRequests, wait for the least
    # recent one to complete.
    while len(results) > numRequests:
        r = results[0]
        del results[0]
        r.waitForCompleted()

# Wait for any remaining requests to complete.
while len(results) > 0:
    r = results[0]
    del results[0]
    r.waitForCompleted()
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Completion Callbacks in Python

The `begin_` method accepts three optional callback arguments that allow you to be notified asynchronously when a request completes. Here are the corresponding methods for the `getName` operation:

Python

```
def begin_getName(self, number, _response=None, _ex=None,
_sent=None, _ctx=None)
```

The value you pass for the response callback (`_response`), the exception callback (`_ex`), or the sent callback (`_sent`) argument must be a *callable object* such as a function or method. The response callback is invoked when the request completes successfully, and the exception callback is invoked when the operation raises an exception. (The sent callback is primarily used for flow control.)

For example, consider the following callbacks for an invocation of the `getName` operation:

Python

```
def getNameCB(name):
    print "Name is: " + name

def failureCB(ex):
    print "Exception is: " + str(ex)
```

The response callback parameters depend on the operation signature. If the operation has a non-`void` return type, the first parameter of the response callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

The exception callback is invoked if the invocation fails because of an Ice run time exception, or if the operation raises a user exception.

To inform the Ice run time that you want to receive callbacks for the completion of the asynchronous call, you pass the callbacks to the `begin_` method:

Python

```
e = EmployeesPrx.checkedCast(...)

e.begin_getName(99, getNameCB, failureCB)
```

Although the signature of an asynchronous proxy method implies that all of the callbacks are optional and therefore can be supplied in any combination, Ice enforces the following semantics at run time:

- If you omit all callbacks, you must call the `end_` method explicitly as described earlier.
- If you supply either a response callback or a sent callback (or both), you must also supply an exception callback.
- You may omit the response callback for an operation that returns no data (that is, an operation with a `void` return type and no out-parameters).

Sharing State Between `begin_` and `end_` Methods in Python

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

Assuming that we have a `Widget` class that designates a particular user interface element, you could pass different widgets by storing the widget to be used as a member of a callback class:

Python

```
class MyCallback(object):
    def __init__(self, w):
        self._w = w

    def getNameCB(self, name):
        self._w.writeString(name)

    def failureCB(self, ex):
        print "Exception is: " + str(ex)
```

For this example, we assume that widgets have a `writeString` method that updates the relevant UI element.

When you call the `begin_` method, you pass the appropriate callback instance to inform the `end_` method how to update the display:

Python

```
e = EmployeesPrx.checkedCast(...)
widget1 = ...
widget2 = ...

# Invoke the getName operation with different widget callbacks.
cb1 = MyCallback(widget1)
e.begin_getName(99, cb1.getNameCB, cb1.failureCB)
cb2 = MyCallback(widget2)
e.begin_getName(24, cb2.getNameCB, cb2.failureCB)
```

The callback class provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same callback instance to multiple invocations. (If you do this, your callback methods may need to use synchronization.)

For those situations in which a stateless callback is preferred, you can use a lambda function to pass state to a callback. Consider the following example:

Python

```
def getNameCB(name, w):
    w.writeString(name)

def failureCB(ex):
    print "Exception is: " + str(ex)

e = EmployeesPrx.checkedCast(...)
widget1 = ...
widget2 = ...

# Use lambda functions to pass state.
e.begin_getName(99, lambda name: getNameCB(name, widget1), failureCB)
e.begin_getName(24, lambda name: getNameCB(name, widget2), failureCB)
```

This strategy eliminates the need to encapsulate shared state in a callback class. Since lambda functions can refer to variables in the enclosing scope, they provide a convenient way to pass state directly to your callback.

Asynchronous Oneway Invocations in Python

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws a `RuntimeError`.

The callback signatures look exactly as for a twoway invocation, but the response method is never called and may be omitted.

Flow Control in Python

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

You can supply a sent callback to be notified when the request was successfully sent:

Python

```
def response(name):
    # ...

def exception(ex):
    # ...

def sent(sentSynchronously):
    # ...
```

You inform the Ice run time that you want to be informed when a call has been passed to the local transport as usual:

Python

```
e.begin_getName(99, response, exception, sent)
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the sent callback from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the sent callback from a different thread once it has written the request to the local transport. The boolean `sentSynchronously` parameter indicates whether the request was sent synchronously or was queued.

The sent callback allows you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Asynchronous Batch Requests in Python

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws a `RuntimeError`.

A batch oneway invocation never calls the callbacks unless an error occurs before the request is queued. The returned `AsyncResult` for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The returned result can also be marked completed if an error occurs before the request is queued.

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency Semantics for AMI in Python

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the sent callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` parameter.

See Also

- Python Mapping for Operations
- Request Contexts
- Batched Invocations

Code Generation in Python

The Python mapping supports two forms of code generation: dynamic and static.

On this page:

Dynamic Code Generation in Python

Using dynamic code generation, Slice files are "loaded" at run time and dynamically translated into Python code, which is immediately compiled and available for use by the application. This is accomplished using the `Ice.loadSlice` function, as shown in the following example:

Python

```
Ice.loadSlice("Color.ice")
import M

print("My favorite color is " + str(M.Color.blue))
```

For this example, we assume that `Color.ice` contains the following definitions:

Slice

```
module M
{
    enum Color { red, green, blue }
}
```

The code imports module `M` after the Slice file is loaded because module `M` is not defined until the Slice definitions have been translated into Python.

`Ice.loadSlice` Options in Python

The `Ice.loadSlice` function behaves like a Slice compiler in that it accepts command-line arguments for specifying preprocessor options and controlling code generation. The arguments must include at least one Slice file.

The function has the following Python definition:

Python

```
def Ice.loadSlice(cmd, args=[])
```

The command-line arguments can be specified entirely in the first argument, `cmd`, which must be a string. The optional second argument can be used to pass additional command-line arguments as a list; this is useful when the caller already has the arguments in list form. The function always returns `None`.

For example, the following calls to `Ice.loadSlice` are functionally equivalent:

Python

```
Ice.loadSlice("-I/opt/IcePy/slice Color.ice")
Ice.loadSlice("-I/opt/IcePy/slice", ["Color.ice"])
Ice.loadSlice("", ["-I/opt/IcePy/slice", "Color.ice"])
```

In addition to the standard compiler options, `Ice.loadSlice` also supports the following command-line options:

- `--all`
  Generate code for all Slice definitions, including those from included files.
- `--checksum`
  Generate checksums for Slice definitions.

Locating Slice Files in Python

If your Slice files depend on Ice types, you can avoid hard-coding the path name of your Ice installation directory by calling the `Ice.getSliceDir` function:

Python

```
Ice.loadSlice("-I" + Ice.getSliceDir() + " Color.ice")
```

This function attempts to locate the `slice` subdirectory of your Ice installation using an algorithm that succeeds for the following scenarios:

- Installation of a binary Ice archive
- Installation of an Ice source distribution using `make install`
- Installation via a Windows installer
- Package installation on Linux (DEB/RPM)
- Execution inside a compiled Ice source distribution

If the `slice` subdirectory can be found, `getSliceDir` returns its absolute path name, otherwise the function returns `None`.

Loading Multiple Slice Files in Python

You can specify as many Slice files as necessary in a single invocation of `Ice.loadSlice`, as shown below:

Python

```
Ice.loadSlice("Syscall.ice Process.ice")
```

Alternatively, you can call `Ice.loadSlice` several times:

Python

```
Ice.loadSlice("Syscall.ice")
Ice.loadSlice("Process.ice")
```

If a Slice file includes another file, the default behavior of `Ice.loadSlice` generates Python code only for the named file. For example, suppose `Syscall.ice` includes `Process.ice` as follows:

Slice

```
// Syscall.ice
#include <Process.ice>
...
```

If you call `Ice.loadSlice("-I. Syscall.ice")`, Python code is not generated for the Slice definitions in `Process.ice` or for any definitions that may be included by `Process.ice`. If you also need code to be generated for included files, one solution is to load them individually in subsequent calls to `Ice.loadSlice`. However, it is much simpler, not to mention more efficient, to use the `--all` option instead:

Python

```
Ice.loadSlice("--all -I. Syscall.ice")
```

When you specify `--all`, `Ice.loadSlice` generates Python code for all Slice definitions included directly or indirectly from the named Slice files.

There is no harm in loading a Slice file multiple times, aside from the additional overhead associated with code generation. For example, this situation could arise when you need to load multiple top-level Slice files that happen to include a common subset of nested files. Suppose that we need to load both `Syscall.ice` and `Kernel.ice`, both of which include `Process.ice`. The simplest way to load both files is with a single call to `Ice.loadSlice`:

Python

```
Ice.loadSlice("--all -I. Syscall.ice Kernel.ice")
```

Although this invocation causes the Ice extension to generate code twice for `Process.ice`, the generated code is structured so that the interpreter ignores duplicate definitions. We could have avoided generating unnecessary code with the following sequence of steps:

Python

```
Ice.loadSlice("--all -I. Syscall.ice")
Ice.loadSlice("-I. Kernel.ice")
```

In more complex cases, however, it can be difficult or impossible to completely avoid this situation, and the overhead of code generation is usually not significant enough to justify such an effort.

Static Code Generation in Python

You should be familiar with static code generation if you have used other Slice language mappings, such as C++ or Java. Using static code

generation, the Slice compiler `slice2py` generates Python code from your Slice definitions.

Compiler Output in Python

For each Slice file `X.ice`, `slice2py` generates Python code into a file named `X_ice.py`.

> Using the file name `X.py` would create problems if `X.ice` defined a module named `X`, therefore the suffix `_ice` is appended to the name of the generated file to prevent name collisions.

The default output directory is the current working directory, but a different directory can be specified using the `--output-dir` option. You can further customize the location of generated files using metadata.

In addition to the generated file, `slice2py` creates a Python package for each Slice module it encounters. A Python package is nothing more than a subdirectory that contains a file with a special name (`__init__.py`). This file is executed automatically by Python when a program first imports the package. It is created by `slice2py` and must not be edited manually. Inside the file is Python code to import the generated files that contain definitions in the Slice module of interest.

For example, the Slice files `Process.ice` and `Syscall.ice` both define types in the Slice module `OS`. First we present `Process.ice`:

Slice

```
module OS
{
    interface Process
    {
        void kill();
    }
}
```

And here is `Syscall.ice`:

Slice

```
#include <Process.ice>
module OS
{
    interface Syscall
    {
        Process getProcess(int pid);
    }
}
```

Next, we translate these files using the Slice compiler:

```
> slice2py -I. Process.ice Syscall.ice
```

If we list the contents of the output directory, we see the following entries:

Python

```
OS/
OS/__init__.py
Process_ice.py
Syscall_ice.py
```

The subdirectory `OS` is the Python package that `slice2py` created for the Slice module `OS`. Inside this directory is the special file `__init__.py` that contains the following statements:

Python

```
import Process_ice
import Syscall_ice
```

Now when a Python program executes `import OS`, the two files `Process_ice.py` and `Syscall_ice.py` are implicitly imported.

Subsequent invocations of `slice2py` for Slice files that also contain definitions in the `OS` module result in additional `import` statements being added to `OS/__init__.py`. Be aware, however, that `import` statements may persist in `__init__.py` files after a Slice file is renamed or becomes obsolete. This situation may manifest itself as a run-time error if the interpreter can no longer locate the generated file while attempting to import the package. It may also cause more subtle problems, if an obsolete generated file is still present and being loaded unintentionally. In general, it is advisable to remove the package directory and regenerate it whenever the set of Slice files changes.

A Python program may also import a generated file explicitly, using a statement such as `import Process_ice`. Typically, however, it is more convenient to import the Python module once, rather than importing potentially several individual files that comprise the module, especially when you consider that the program must still import the module explicitly in order to make its definitions available. For example, it is much simpler to state

Python

```
import OS
```

rather than the following alternative:

Python

```
import Process_ice
import Syscall_ice
import OS
```

Customizing Compiler Output using Metadata in Python

As we showed in the previous section, the compiler by default generates all files and package directories into the specified output directory. Furthermore, the generated code assumes that all of the generated files are globally accessible via the Python search path. Continuing with our last example, the package initialization file `OS/__init__.py` will contain the statements:

Python

```
import Process_ice
import Syscall_ice
```

An application that wants to import these Slice definitions must therefore ensure that its search path includes the directory containing `Process_ice.py`, `Syscall_ice.py`, and the `OS` subdirectory.

For an application with many Slice files, the output directory can quickly become cluttered with generated `*_ice.py` files. If you intend to install the compiler output into a common system directory such as `site-packages`, you may want more control over the location of the files. Note that you can't simply move the `*_ice.py` files into a different subdirectory without also adding that directory to the search path or modifying every generated file that imports those files.

A simpler solution is to add a global metadata directive named `python:pkgdir` to each Slice file that specifies the directory into which the compiler should place the corresponding `*_ice.py` file. This directive also affects the `import` statement that the compiler emits whenever the generated code is referenced. The compiler treats the specified directory as being relative to the output directory denoted by `--output-dir`, or to the current working directory if `--output-dir` is not defined.

Let's add the metadata directive to our sample Slice files, starting with `Process.ice`:

Slice

```
[["python:pkgdir:OS"]]

module OS
{
    interface Process
    {
        void kill();
    }
}
```

And here is `Syscall.ice`:

Slice

```
[["python:pkgdir:OS"]]

#include <Process.ice>
module OS
{
    interface Syscall
    {
        Process getProcess(int pid);
    }
}
```

We used the same directive, `python:pkgdir:OS`, in both Slice files. The compiler now generates the following output:

Python

```
OS/
OS/__init__.py
OS/Process_ice.py
OS/Syscall_ice.py
```

The package initialization file `OS/__init__.py` imports the `*_ice.py` files as shown below:

Python

```
import OS.Process_ice
import OS.Syscall_ice
```

Our metadata example specifies the same directory as the top-level Slice module `OS`, which means the `*_ice.py` files are placed into the top-level Python package `OS`. This is convenient from an organizational standpoint but you are not required to use the same name. We could specify an arbitrary name, such as `python:pkgdir:OS_gen`, and the compiler would generate:

Python

```
OS/
OS/__init__.py
OS_gen/__init__.py    # This file is empty
OS_gen/Process_ice.py
OS_gen/Syscall_ice.py
```

Using an alternate directory as shown above does not affect the package in which the definitions are placed at run time. For example, your application would still refer to `OS.Process` as before. The `python:pkgdir` metadata only affects the directory in which the `*_ice.py` file is placed.

The metadata directive can optionally specify a nested subdirectory, such as `python:pkgdir:OS/gen`. The compiler would generate `OS/gen/Process_ice.py` into the output directory, and this file would be imported as `OS.gen.Process_ice`. Your metadata directive must use forward slashes when specifying a nested subdirectory.

Include Files in Python

It's important to understand how `slice2py` handles include files. In the absence of the `--all` option, the compiler does not generate Python code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into Python `import` statements as described below.

Include Files with `python:pkgdir` Metadata

When the include file contains a `python:pkgdir` metadata directive, the specified directory is translated into a package name. For example, the metadata directive `python:pkgdir:OS/gen` becomes the package name `OS.gen`. If the name of the include file is `Process.ice`, then the compiler generates

```
import OS.gen.Process_ice
```

Include Files without `python:pkgdir` Metadata

If the `python:pkgdir` metadata is not present, the compiler determines the full path name of the include file and creates the shortest possible relative path name for the include file by iterating over each of the include directories (specified using the `-I` option) and removing the leading directory from the include file if possible. For example, if the full path name of an include file is `/opt/App/slice/OS/Process.ice`, and we specified the options `-I/opt/App` and `-I/opt/App/slice`, then the shortest relative path name is `OS/Process.ice` after

removing `/opt/App/slice`. Any remaining slashes are replaced with underscores, so the `import` statement for `OS/Process.ice` becomes

```
import OS_Process_ice
```

There is a potential problem here that must be addressed. The generated `import` statement for our example above expects to find the file `OS_Process_ice.py` somewhere in Python's search path. However, `slice2py` uses a different default name, `Process_ice.py`, when it compiles `Process.ice.`. To resolve this issue, we must use the `--prefix` option when compiling `Process.ice`:

```
slice2py --prefix OS_ Process.ice
```

The `--prefix` option causes the compiler to prepend the specified prefix to the name of each generated file. When executed, the above command creates the desired file name: `OS_Process_ice.py.`

It should be apparent by now that generating Python code for a complex Ice application requires a bit of planning. In particular, it is imperative that you be consistent in your use of `#include` statements, include directories, and `--prefix` options to ensure that the correct file names are used at all times.

Of course, these precautionary steps are only necessary when you are compiling Slice files individually. An alternative is to use the `--all` option and generate Python code for all of your Slice definitions into one Python source file. If you do not have a suitable Slice file that includes all necessary Slice definitions, you could write a "master" Slice file specifically for this purpose.

Static Versus Dynamic Code Generation in Python

There are several issues to consider when evaluating your requirements for code generation.

Application Considerations for Code Generation in Python

The requirements of your application generally dictate whether you should use dynamic or static code generation. Dynamic code generation is convenient for a number of reasons:

- it avoids the intermediate compilation step required by static code generation
- it makes the application more compact because the application requires only the Slice files, not the assortment of files and directories produced by static code generation
- it reduces complexity, which is especially helpful during testing, or when writing short or transient programs.

Static code generation, on the other hand, is appropriate in many situations:

- when an application uses a large number of Slice definitions and the startup delay must be minimized
- when it is not feasible to deploy Slice files with the application
- when a number of applications share the same Slice files
- when Python code is required in order to utilize third-party Python tools.

Mixing Static and Dynamic Code Generation in Python

Using a combination of static and dynamic translation in an application can produce unexpected results. For example, consider a situation where a dynamically-translated Slice file includes another Slice file that was statically translated:

Slice

```
// Slice
#include <Glacier2/Session.ice>

module App
{
    interface SessionFactory
    {
        Glacier2::Session* createSession();
    }
}
```

The Slice file `Session.ice` is statically translated, as are all of the Slice files included with the Ice run time.

Assuming the above definitions are saved in `App.ice`, let's execute a simple Python script:

Python

```
# Python
import Ice
Ice.loadSlice("-I/opt/Ice/slice App.ice")

import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier): # Error
    def checkPermissions(self, userId, password):
        return (True, "")
```

The code looks reasonable, but running it produces the following error:

```
'module' object has no attribute 'PermissionsVerifier'
```

Normally, importing the Glacier2 module as we have done here would load all of the Python code generated for the Glacier2 Slice files. However, since `App.ice` has already included a subset of the Glacier2 definitions, the Python interpreter ignores any subsequent requests to import the entire module, and therefore the `PermissionsVerifier` type is not present.

One way to address this problem is to import the statically-translated modules first, prior to loading Slice files dynamically:

Python

```
# Python
import Ice, Glacier2 # Import Glacier2 before App.ice is loaded
Ice.loadSlice("-I/opt/Ice/slice App.ice")

class MyVerifier(Glacier2.PermissionsVerifier): # OK
    def checkPermissions(self, userId, password):
        return (True, "")
```

The disadvantage of this approach in a non-trivial application is that it breaks encapsulation, forcing one Python module to know what other modules are doing. For example, suppose we place our `PermissionsVerifier` implementation in a module named `verifier.py`:

Python

```
# Python
import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier):
    def checkPermissions(self, userId, password):
        return (True, "")
```

Now that the use of Glacier2 definitions is encapsulated in `verifier.py`, we would like to remove references to Glacier2 from the main script:

Python

```
# Python
import Ice
Ice.loadSlice("-I/opt/Ice/slice App.ice")
...
import verifier # Error
v = verifier.MyVerifier()
```

Unfortunately, executing this script produces the same error as before. To fix it, we have to break the `verifier` module's encapsulation and import the `Glacier2` module in the main script because we know that the `verifier` module requires it:

Python

```
# Python
import Ice, Glacier2
Ice.loadSlice("-I/opt/Ice/slice App.ice")
...
import verifier # OK
v = verifier.MyVerifier()
```

Although breaking encapsulation in this way might offend our sense of good design, it is a relatively minor issue.

Another solution is to import the necessary submodules explicitly. We can safely remove the Glacier2 reference from our main script after rewriting `verifier.py` as shown below:

Python

```
# Python
import Glacier2.PermissionsVerifier_ice
import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier):
    def checkPermissions(self, userId, password):
        return (True, "")
```

Using the rules defined for static code generation, we can derive the name of the module containing the code generated for `PermissionsV`

`erifier.ice` and import it directly. We need a second `import` statement to make the Glacier2 definitions accessible in this module.

`slice2py` Command-Line Options

The Slice-to-Python compiler, `slice2py`, offers the following command-line options in addition to the standard options:

- `--all`
  Generate code for all Slice definitions, including those from included files.
- `--checksum`
  Generate checksums for Slice definitions.
- `--prefix PREFIX`
  Use *`PREFIX`* as the prefix for generated file names.

Generating Packages in Python

By default, the scope of a Slice definition determines the module of its mapped Python construct. There are times, however, when applications require greater control over the packaging of generated Python code. For example, consider the following Slice definitions:

Slice

```
module sys
{
    interface Process
    {
        // ...
    }
}
```

Other language mappings can use these Slice definitions as shown, but they present a problem for the Python mapping: the top-level Slice module `sys` conflicts with Python's predefined module sys. A Python application executing the statement `import sys` would import whichever module the interpreter happens to locate first in its search path.

A workaround for this problem is to modify the Slice definitions so that the top-level module no longer conflicts with a predefined Python module, but that may not be feasible in certain situations. For example, the application may already be deployed using other language mappings, in which case the impact of modifying the Slice definitions could represent an unacceptable expense.

The Python mapping could have addressed this issue by considering the names of predefined modules to be reserved, in which case the Slice module `sys` would be mapped to the Python module `_sys`. However, the likelihood of a name conflict is relatively low to justify such a solution, therefore the mapping supports a different approach: metadata can be used to enclose generated code in a Python package. Our modified Slice definitions demonstrate this feature:

Slice

```
["python:package:zeroc"]
module sys
{
    interface Process
    {
        // ...
    }
}
```

The metadata directive `python:package:zeroc` causes the mapping to generate all of the code resulting from definitions in module `sys` into the Python package `zeroc`. The net effect is the same as if we had enclosed our Slice definitions in the module `zeroc`: the Slice module `sys` is mapped to the Python module `zeroc.sys`. However, by using metadata we have not affected the semantics of the Slice definitions, nor have we affected other language mappings.

> The `python:package` directive can also be applied as global metadata, in which case it serves as the default directive unless overridden by module metadata.

See Also

- Using the Slice Compilers
- Python Mapping for Modules
- Using Slice Checksums in Python
- Metadata
- Slice Metadata Directives

Using Slice Checksums in Python

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2py`, the `--checksum` option causes the compiler to generate code that adds checksums to the dictionary `Ice.sliceChecksums`. The checksums are installed automatically when the Python code is first imported; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the dictionaries using the comparison operator. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

Python

```
serverChecksums = ...
for i in Ice.sliceChecksums:
    if not serverChecksums.has_key(i):
        # No match found for type id!
    elif Ice.sliceChecksums[i] != serverChecksums[i]:
        # Checksum mismatch!
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums

Example of a File System Client in Python

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The Python code shown here hardly differs from the code you would write for an ordinary Python program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Python object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side Python mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

Python

```
import signal, sys, Ice

Ice.loadSlice('Filesystem.ice')
import Filesystem

# Recursively display the contents of directory "dir"
# in tree fashion. For files, show the contents of
# each file. The "depth" parameter is the current
# nesting level (for indentation).

def listRecursive(dir, depth):
    indent = ''
    depth = depth + 1
    for i in range(depth):
        indent = indent + '\t'

    contents = dir.list()

    for node in contents:
        subdir = Filesystem.DirectoryPrx.checkedCast(node)
        file = Filesystem.FilePrx.uncheckedCast(node)
        sys.stdout.write(indent + node.name() + " ")
        if subdir:
            print("(directory):")
            listRecursive(subdir, depth)
        else:
            print("(file):")
            text = file.read()
            for line in text:
                print(indent + "\t" + line)

with Ice.initialize(sys.argv) as communicator:
    #
    # Create a proxy to the root directory
    #
    obj = communicator.stringToProxy("RootDir:default -h localhost -p
10000")

    #
    # Downcast the proxy to a Directory proxy
    #
    rootDir = Filesystem.DirectoryPrx.checkedCast(obj)

    #
    # Recursively list the contents of the root directory
    #
    print("Contents of root directory:")
    listRecursive(rootDir, 0)
```

The program first defines the `listRecursive` function, which is a helper function to print the contents of the file system, and the main

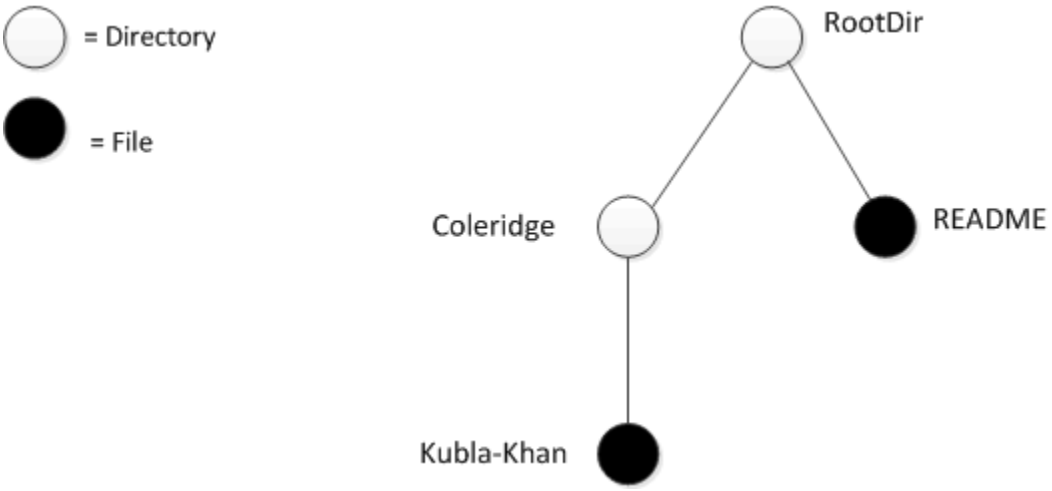
program follows. Let us look at the main program first:

1. The structure of the code follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of a two files and a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Example of a File System Server in Python
- Object Life Cycle
- IceGrid

Server-Side Slice-to-Python Mapping

The mapping for Slice data types to Python is identical on the client side and server side. This means that everything in the Client-Side Slice-to-Python Mapping section also applies to the server side. However, for the server side, there are a few additional things you need to know — specifically, how to:

- Implement servants
- Pass parameters and throw exceptions
- Create servants and register them with the Ice run time.

Although the examples in this chapter are simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers, you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Python mapping rules beyond those described here.

> The Python interpreter's Global Interpreter Lock (GIL) can impact an Ice server's ability to utilize multiple processing cores. Refer to our FAQ for for more information.

Topics

Server-Side Python Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing methods in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in Python

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton classes*. A skeleton is an abstract base class from which you derive your servant class and define a method for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The Python mapping generates the following definition for this interface:

Python

```
class Node(Ice.Object):
    def __init__(self):
        # ...

    #
    # Operation signatures.
    #
    # def name(self, current=None):
```

The important points to note here are:

- As for the client side, Slice modules are mapped to Python modules with the same name, so the skeleton class definitions are part of the `Filesystem` module.
- The name of the skeleton class is the same as the Slice interface (`Node`).
- The skeleton class contains a comment summarizing the method signature of each operation in the Slice interface.
- The skeleton class is an abstract base class because its constructor prevents direct instantiation of the class.
- The skeleton class inherits from `Ice.Object` (which forms the root of the Ice object hierarchy).

`Ice.Object` Base Class for Python Servants

`Object` is mapped to the `Ice.Object` class in Python:

Python

```
class Object(object):
    def ice_isA(self, id, current=None):
        # ...
    def ice_ping(self, current=None):
        # ...
    def ice_ids(self, current=None):
        # ...
    def ice_id(self, current=None):
        # ...

    @staticmethod
    def ice_staticId():
        # ...
```

The methods of `Ice.Object` behave as follows:

- `ice_isA`
  This method returns `true` if target object implements the given type ID, and `false` otherwise.
- `ice_ping`
  `ice_ping` provides a basic reachability test for the servant.
- `ice_ids`
  This method returns a string array representing all of the type IDs implemented by this servant, including `::Ice::Object`.
- `ice_id`
  This method returns the type ID of the most-derived interface implemented by this servant.
- `ice_staticID`
  This static method returns the type ID of the target class: `::Ice::Object` when called on `Ice.Object`.

Servant Classes in Python

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

Python

```
import Filesystem

class NodeI(Filesystem.Node):
    def __init__(self, name):
        self._name = name

    def name(self, current=None):
        return self._name
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.) Note that `NodeI` extends `Filesystem._NodeDisp`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that is defined in the `Node` interface. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` function returns its value.)

Server-Side Normal and `idempotent` Operations in Python

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

Slice

```
interface Example
{
                void normalOp();
    idempotent void idempotentOp();
}
```

The mapping for this interface is shown below:

Python

```
class Example(Ice.Object):
    # ...

    #
    # Operation signatures.
    #
    # def normalOp(self, current=None):
    # def idempotentOp(self, current=None):
```

Note that the signatures of the methods are unaffected by the `idempotent` qualifier.

See Also

- Slice for a Simple File System
- Python Mapping for Interfaces
- Parameter Passing in Python
- Raising Exceptions in Python

Parameter Passing in Python

Parameter passing on the server side follows the rules for the client side. Additionally, every operation has a trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method in a Python servant has a `current` parameter. We will ignore this parameter for now.

On this page:

Server-Side Mapping for Parameters in Python

For each `in` parameter of a Slice operation, the Python mapping generates a corresponding parameter for the method in the skeleton. An operation returning multiple values returns them in a tuple consisting of a non-`void` return value, if any, followed by the `out` parameters in the order of declaration. An operation returning only one value simply returns the value itself.

> An operation returns multiple values when it declares multiple out parameters, or when it declares a non-`void` return type and at least one out parameter.

To illustrate these rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
interface Example
{
    string op1(string sin);
    void op2(string sin, out string sout);
    string op3(string sin, out string sout);
}
```

The generated skeleton class for this interface looks as follows:

Python

```
class Example(Ice.Object):
    def __init__(self):
        # ...

    #
    # Operation signatures.
    #
    # def op1(self, sin, current=None):
    # def op2(self, sin, current=None):
    # def op3(self, sin, current=None):
```

The signatures of the Python methods are identical because they all accept a single in parameter, but their implementations differ in the way they return values. For example, we could implement the operations as follows:

Python

```
class ExampleI(Example):
    def op1(self, sin, current=None):
        print sin             # In params are initialized
        return "Done"         # Return value

    def op2(self, sin, current=None):
        print sin             # In params are initialized
        return "Hello World!" # Out parameter

    def op3(self, sin, current=None):
        print sin             # In params are initialized
        return ("Done", "Hello World!")
```

Notice that `op1` and `op2` return their string values directly, whereas `op3` returns a tuple consisting of the return value followed by the out parameter.

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal Python rules and do not require special-purpose API calls.

Thread-Safe Marshaling in Python

The marshaling semantics of the Ice run time and the Python interpreter present a subtle thread safety issue that arises when an operation returns data by reference. For Python applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

> In the C-based implementation of Python ("Cython"), only one thread at a time can be executing in the interpreter. However, depending on your thread pool configuration, there may be multiple Ice threads waiting to enter the interpreter. You should write your code to assume that the interpreter can switch to a different thread at any time.

The potential for corruption occurs whenever a servant returns an instance of one of these types, yet continues to hold a reference to that data. For example, consider the following servant implementation:

Python

```
class GridI(Grid):
    def __init__(self):
        self._grid = # ...

    def getGrid(self, current):
        return self._grid

    def setValue(self, x, y, val, current):
        self._grid[x][y] = val
```

Suppose that a client invoked the `getGrid` operation, and another client invoked the `setValue` operation. The interpreter allows a thread to dispatch the call to `getGrid`, but before control returns to the Ice run time, the interpreter switches threads to allow the call to `setValue` to proceed. The problem is that `setValue` can modify the data before the thread that invoked `getGrid` has a chance to marshal its results.

In most cases this won't cause a failure, but it does mean that an invocation might return different results than it intended. Furthermore, adding synchronization to the `getGrid` and `setValue` operations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

Python

```
class GridI(Grid):
    def __init__(self):
        self._lock = threading.Lock()

    def getGrid(self, current):
        with self._lock:
            return self._grid

    def setValue(self, x, y, val, current):
        with self._lock:
            newGrid = # shallow copy...
            newGrid[x][y] = val
            self._grid = newGrid
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `marshaled-result` metadata causes additional code to be generated, but only if that operation returns one or more of the mutable types listed earlier. The metadata directive has the following effects:

- For an operation op that returns at least one mutable type, the Slice compiler generates a static method named `OpMarshaledResult`. This method takes two parameters: the result value (or result tuple, if the operation returns multiple values), and a `Current`. The method marshals the results immediately, and the servant must supply the `Current` in order for the results to be marshaled correctly. Your servant must return the result of this method as its return value.
- A servant method can still optionally return its results using the regular mapping instead, as if the `marshaled-result` metadata was not present. Use caution to ensure no unexpected behavior can occur.

The metadata directive has no effect on the proxy mapping, nor does it generate a `MarshaledResult` method for Slice operations that return `void` or return only immutable values.

> You can also annotate an interface with the `marshaled-result` metadata and it will be applied to all of the interface's operations.

After applying the metadata, we can now implement the `Grid` servant as follows:

Python

```
class GridI(Grid):
    def __init__(self):
        self._lock = threading.Lock()

    def getGrid(self, current):
        with self._lock:
            return Grid.GetGridMarshaledResult(self._grid, current) #
_grid is marshaled immediately

    def setValue(self, x, y, val, current):
        with self._lock:
            self._grid[x][y] = val # this is safe
```

See Also

- Server-Side Python Mapping for Interfaces
- Python Mapping for Operations
- Raising Exceptions in Python
- The Current Object

Raising Exceptions in Python

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

Python

```
class FileI(Filesystem.File):
    # ...

    def write(self, text, current=None):
        # Try to write the file contents here...
        # Assume we are out of space...
        if error:
            e = Filesystem.GenericError()
            e.reason = "file too large"
            raise e
```

The mapping for exceptions generates a constructor that accepts values for data members, so we can simplify this example by changing our `raise` statement to the following:

Python

```
class FileI(Filesystem.File):
    # ...

    def write(self, text, current=None):
        # Try to write the file contents here...
        # Assume we are out of space...
        if error:
            raise Filesystem.GenericError("file too large")
```

If you throw an arbitrary Python run-time exception, the Ice run time catches the exception and then returns an `UnknownException` to the client.

The server-side Ice run time does not validate user exceptions thrown by an operation implementation to ensure they are compatible with the operation's Slice definition. Rather, Ice returns the user exception to the client, where the client-side run time will validate the exception as usual and raise `UnknownUserException` for an unexpected exception type.

If you throw an Ice run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`. For that reason, you should never throw Ice run-time exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

> Three run-time exceptions are treated specially and not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`.

See Also

- Run-Time Exceptions
- Server-Side Python Mapping for Interfaces
- Python Mapping for Exceptions
- Versioning

Object Incarnation in Python

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a Python Servant

Instantiating a servant means to allocate an instance:

Python

```
servant = NodeI("Fred")
```

This statement creates a new `NodeI` instance and assigns its reference to the variable `servant`.

Creating an Identity in Python

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

> The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
    // ...
}
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant

locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

Python

```
id = Ice.Identity()
id.name = "Fred" # Not unique, but good enough for now
```

Note that the mapping for structures allows us to write the following equivalent code:

Python

```
id = Ice.Identity("Fred") # Not unique, but good enough for now
```

Activating a Python Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `adapter` variable, we can write:

Python

```
adapter.add(servant, id)
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

UUIDs as Identities in Python

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. The `Ice.generateUUID` function creates such identities:

Python

```
import Ice
print Ice.generateUUID()
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` c reates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

Python

```
adapter.addWithUUID(NodeI("Fred"))
```

Creating Proxies in Python

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Hello World Application. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in Python

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

Python

```
proxy = adapter.addWithUUID(NodeI("Fred"))
nodeProxy = Filesystem.NodePrx.uncheckedCast(proxy)

# Pass nodeProxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

Direct Proxy Creation in Python

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        // ...
    }
}
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

Python

```
id = Ice.Identity()
id.name = Ice.generateUUID()
proxy = adapter.createProxy(id)
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Hello World Application
- Python Mapping for Structures
- Server-Side Python Mapping for Interfaces
- Object Adapter States
- Servant Locators
- Object Life Cycle

Asynchronous Method Dispatch (AMD) in Python

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool. If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server can provide its results to the Ice run time for delivery to the client.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

On this page:

AMD Mapping in Python

Annotating operations with `["amd"]` metadata directives has no effect in the Python mapping. In fact, the mappings for synchronous and asynchronous dispatch are nearly identical, with the only difference being the return type: the operation has asynchronous semantics if you return a future, otherwise the operation has synchronous semantics. The parameter passing rules for in parameters are the same in both cases.

An asynchronous implementation will normally return an instance of `Ice.Future`. However, Ice also accepts any other future type that provides an `add_done_callback` method, such as `asyncio.Future` or `concurrent.futures.Future`. Ice registers its own completion callback with the future so that, upon completion of the invocation, Ice can marshal the results or exception.

> The implementation is responsible for ensuring that all futures complete successfully or exceptionally. Neglecting to complete a future can cause the client's invocation to hang indefinitely.

Consider the following operation:

Slice

```
interface Test
{
    int foo(short s, out long l);
}
```

We can implement operation `foo` as follows:

Python

```
class TestI(Test):
    def foo(s, current=None):
        if s < 5:
            return (1, 2) # Synchronous dispatch
        else:
            f = Ice.Future()
            # Asynchronous dispatch, e.g., queue the request, start a
thread, etc.
            # We eventually need to complete this future, such as with
            # f.set_result((1, 2))
            return f
```

Unlike previous versions of the Python mapping, the name of the dispatch method does not use an AMD-specific suffix; the method name is the same regardless of whether you intend to use synchronous or asynchronous dispatch. The implementation can also use both styles, as shown in the example above. If the implementation returns something other than a future, including `None` for an operation that returns no values, Ice assumes the operation has completed successfully and marshals the results immediately.

The `Ice.Future` class accepts a single value as its result. If a Slice operation returns multiple values, including the return value and all out parameters, they must be supplied as a tuple to `set_result`. The semantics are identical to those for synchronous dispatch.

AMD Thread Safety in Python

As with the synchronous mapping, you can add the `marshaled-result` metadata to operations that return mutable types in order to avoid potential thread-safety issues. Your asynchronous operation can then return a future whose result is *Op*`MarshaledResult`.

AMD Exceptions in Python

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).

> These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

Although we recommend that the future be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the future. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

AMD Example in Python

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

Slice

```
module Demo
{
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {}

    interface Model
    {
        Grid interpolate(Grid data, float factor)
            throws RangeError;
    }
}
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo.Model` and supplies a definition for the `interpolate` method that creates a `Job` to hold the future and arguments, and adds the `Job` to a queue. The method uses a lock to guard access to the queue:

Python

```
class ModelI(Demo.Model):
    def __init__(self):
        self._mutex = threading.Lock()
        self._jobs = []

    def interpolate(self, data, factor, current=None):
        with self._mutex:
            f = Ice.Future()
            self._jobs.append(Job(f, data, factor))
            return f
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

Python

```
class Job(object):
    def __init__(self, future, grid, factor):
        self._future = future
        self._grid = grid
        self._factor = factor

    def execute(self):
        if not self.interpolateGrid():
            self._future.set_exception(Demo.RangeError())
            return
        self._future.set_result(self._grid)

    def interpolateGrid(self):
        # ...
```

If `interpolateGrid` returns false, then we complete the future using `set_exception` to indicate that a range error has occurred. If interpolation was successful, we send the modified grid back to the client by calling `set_result` on the future.

Chaining Asynchronous Invocations in Python

Given that asynchronous proxy invocations return futures, and asynchronous dispatch methods return futures, it becomes quite easy to chain together a sequence of calls under the right circumstances. Specifically, the operations being chained must have the same result types and compatible user exception specifications.

Continuing our example from the previous section, suppose our `Model` implementation delegates its requests to an internal server:

Python

```
class ModelI(Demo.Model):
    def __init__(self, internalModel):
        self._internalModel = internalModel

    def interpolate(self, data, factor, current=None):
        return self._internalModel.interpolateAsync(data, factor)
```

The constructor receives a proxy for the internal model server, and the implementation of `interpolate` simply returns the future created by the asynchronous proxy invocation.

Using Coroutines in Python

For applications using Python 3.5 or later, a Slice operation may optionally be implemented as a coroutine, allowing you to use the `await` keyword to suspend processing while waiting for subtasks to complete. Again using our interpolation example, suppose we need to process the data grid in stages:

Python

```
class ModelI(Demo.Model):
    def __init__(self, internalModel):
        self._internalModel = internalModel

    # Implement as a coroutine
    async def interpolate(self, data, factor, current=None):
        # Stage 1
        data = await self._internalModel.interpolateAsync(data, factor)
        # Stage 2 with new factor
        return await self._internalModel.interpolateAsync(data, factor *
2)
```

Ice automatically detects a dispatch method that is implemented as a coroutine and ensures it runs to completion. A coroutine may await on futures; Ice restarts the coroutine when the future completes and passes the future's result.

See Also

- User Exceptions
- Run-Time Exceptions
- Asynchronous Method Invocation (AMI) in Python
- The Ice Threading Model

Example of a File System Server in Python

This page presents the source code for a Python server that implements our file system and communicates with the client we wrote earlier.

The server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same as a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

On this page:

Implementing a File System Server in Python

We have now seen enough of the server-side Python mapping to implement a server for our file system. (You may find it useful to review these Slice definitions before studying the source code.)

Our server is implemented in a single source file, `server.py`, containing our server's main program as well as the definitions of our `Directory` and `File` servant subclasses.

Server Main Program in Python

Our server main program creates and destroys an Ice communicator and call the `run` helper function. `run` uses this communicator instantiate our file system objects:

Python

```
import signal, sys, Ice

Ice.loadSlice('Filesystem.ice')
import Filesystem

class DirectoryI(Filesystem.Directory):
    # ...

class FileI(Filesystem.File):
    # ...

def run(communicator):
    #
    # Create an object adapter
    #
    adapter = communicator.createObjectAdapterWithEndpoints(
        "SimpleFileSystem", "default -h localhost -p 10000")

    #
    # Create the root directory (with name "/" and no parent)
    #
    root = DirectoryI(communicator, "/", None)
    root.activate(adapter)
```

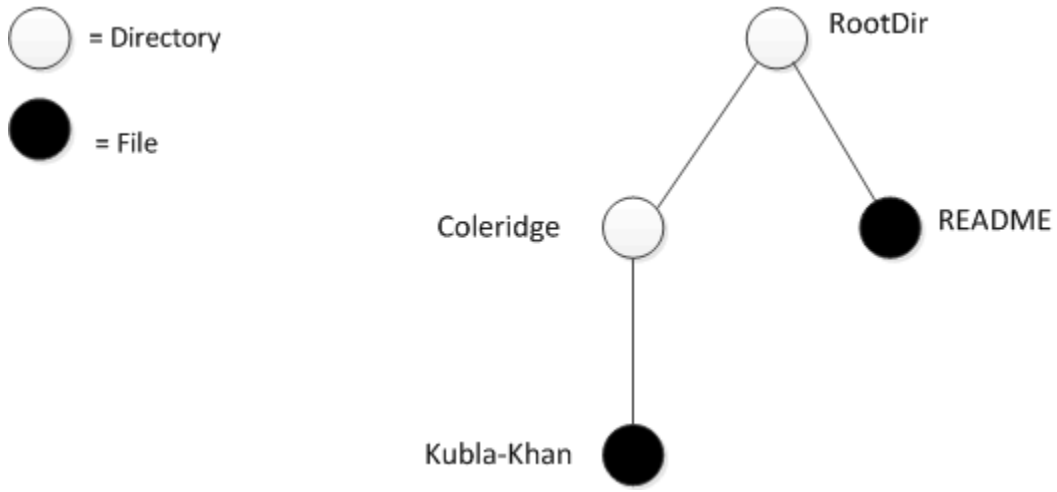
```
    #
    # Create a file called "README" in the root directory
    #
    file = FileI(communicator, "README", root)
    text = ["This file system contains a collection of poetry."]
    try:
        file.write(text, None)
    except Filesystem.GenericError as e:
        print(e.reason)
    file.activate(adapter)

    #
    # Create a directory called "Coleridge" in the root directory
    #
    coleridge = DirectoryI(communicator, "Coleridge", root)
    coleridge.activate(adapter)

    #
    # Create a file called "Kubla_Khan" in the Coleridge directory
    #
    file = FileI(communicator, "Kubla_Khan", coleridge)
    text = ["In Xanadu did Kubla Khan",
            "A stately pleasure-dome decree:",
            "Where Alph, the sacred river, ran",
            "Through caverns measureless to man",
            "Down to a sunless sea."]
    try:
        file.write(text, None)
    except Filesystem.GenericError as e:
        print(e.reason)
    file.activate(adapter)

    #
    # All objects are created, allow client requests now
    #
    adapter.activate()

    #
    # Wait until we are done
    #
    communicator.waitForShutdown()

#
# Ice.initialize returns an initialized Ice communicator,
# the communicator is destroyed once it goes out of scope.
#
with Ice.initialize(sys.argv) as communicator:
    #
    # Install a signal handler to shutdown the communicator on Ctrl-C
```

```
        #
        signal.signal(signal.SIGINT, lambda signum, frame:
```

```
communicator.shutdown())

    run(communicator)
```

Much of this code is boiler plate: we create a communicator, then an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown below:



A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass `None`.) Thus, the statement

Python

```
root = DirectoryI("/", None)
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in the above illustration:

Python

```
# Create the root directory (with name "/" and no parent)
#
root = DirectoryI("/", None)

# Create a file called "README" in the root directory
#
file = FileI("README", root)
text = [ "This file system contains a collection of poetry." ]
try:
    file.write(text)
except Filesystem.GenericError, e:
    print e.reason

# Create a directory called "Coleridge"
# in the root directory
#
coleridge = DirectoryI("Coleridge", root)

# Create a file called "Kubla_Khan"
# in the Coleridge directory
#
file = FileI("Kubla_Khan", coleridge)
text = [ "In Xanadu did Kubla Khan",
         "A stately pleasure-dome decree:",
         "Where Alph, the sacred river, ran",
         "Through caverns measureless to man",
         "Down to a sunless sea." ]
try:
    file.write(text)
except Filesystem.GenericError, e:
    print e.reason
```

We first create the root directory and a file `README` within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

Python

```
text = [ "This file system contains a collection of poetry." ]
try:
    file.write(text)
except Filesystem.GenericError, e:
    print e.reason
```

Recall that Slice sequences map to Python lists. The Slice type `Lines` is simply a list of strings; we add a line of text to our `README` file by initializing the `text` list to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

Python

```
file.write(text)
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and not via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place — such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Python method call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in the above illustration.

`FileI` Servant Class in Python

Our `FileI` servant class has the following basic structure:

Python

```
class FileI(Filesystem.File):
    # Constructor and operations here...

    _adapter = None
```

The class has a number of data members:

- `_adapter`
  This class member stores a reference to the single object adapter we use in our server.
- `_name`
  This instance member stores the name of the file incarnated by the servant.
- `_parent`
  This instance member stores the reference to the servant for the file's parent directory.
- `_lines`
  This instance member holds the contents of the file.

The `_name`, `_parent`, and `_lines` data members are initialized by the constructor:

Python

```
    def __init__(self, name, parent):
        self._name = name
        self._parent = parent
        self._lines = []

        assert(self._parent != None)

        # Create an identity
        #
        myID = Ice.Identity()
        myID.name = Ice.generateUUID()

        # Add ourself to the object adapter
        #
        self._adapter.add(self, myID)

        # Create a proxy for the new node and
        # add it as a child to the parent
        #
        thisNode = Filesystem.NodePrx.uncheckedCast(self._adapter.creat
eProxy(myID))
        self._parent.addChild(thisNode)
```

After initializing the instance members, the code verifies that the reference to the parent is not `None` because every file must have a parent directory. The constructor then generates an identity for the file by calling `Ice.generateUUID` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChil d` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function in `DirectoryI` Methods.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

Python

```
    # Slice Node::name() operation

    def name(self, current=None):
        return self._name

    # Slice File::read() operation

    def read(self, current=None):
        return self._lines

    # Slice File::write() operation

    def write(self, text, current=None):
        self._lines = text
```

The `name` method is inherited from the generated `Node` class. It simply returns the value of the `_name` instance member.

The `read` and `write` methods are inherited from the generated `File` class and simply return and set the `_lines` instance member.

`DirectoryI` Servant Class in Python

The `DirectoryI` class has the following basic structure:

Python

```
class DirectoryI(Filesystem.Directory):
    # Constructor and operations here...

    _adapter = None
```

`DirectoryI` Data Members in Python

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds `None`.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

Python

```
    def __init__(self, name, parent):
        self._name = name
        self._parent = parent
        self._contents = []

        # Create an identity. The
        # parent has the fixed identity "RootDir"
        #
        myID = Ice.Identity()
        if(self._parent):
            myID.name = Ice.generateUUID()
        else:
            myID.name = "RootDir"

        # Add ourself to the object adapter
        #
        self._adapter.add(self, myID)

        # Create a proxy for the new node and
        # add it as a child to the parent
        #
        thisNode = Filesystem.NodePrx.uncheckedCast(self._adapter.creat
eProxy(myID))
        if self._parent:
            self._parent.addChild(thisNode)
```

`DirectoryI` Constructor in Python

The constructor creates an identity for the new directory by calling `Ice.generateUUID`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a proxy to itself and passes it to the `addChild` helper function.

`DirectoryI` Methods in Python

`addChild` simply adds the passed reference to the `_contents` list:

Python

```
def addChild(self, child):
    self._contents.append(child)
```

The remainder of the operations, `name` and `list`, are trivial:

Python

```
def name(self, current=None):
    return self._name

def list(self, current=None):
    return self._contents
```

See Also

- Slice for a Simple File System
- Python Mapping for Sequences
- Example of a File System Client in Python

Slice-to-Python Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to Python is identical to the mapping for these constructs without the `local` qualifier. The generated Python code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

Data members on local Slice types (classes, exceptions and structs) are mapped to Python just like the data members of the corresponding non-local Slice construct.

The rest of this section describes the mapping of the remaining local types to Python:

Python Mapping for Local Interfaces

On this page:

Mapped Python Class

A Slice local interface is mapped to a Python class with the same name. For example:

Slice

```
module M
{
    local interface Example
    {
        ...
    }
}
```

is mapped to the Python class `Example`:

Python

```
class Example(object):
    ...
```

LocalObject in Python

All Slice local interfaces implicitly derive from `LocalObject`, which is mapped to the native `object` class in Python.

Mapping for Local Interface Inheritance in Python

Inheritance of local Slice interfaces is mapped to class inheritance in Python. For example:

Slice

```
module M
{
    local interface A {}
    local interface B extends A {}
    local interface C extends A {}
    local interface D extends B, C {}
}
```

is mapped to:

Python

```
class A(object):
    ...
class B(A):
    ...
class C(A):
    ...
class D(B, C):
    ...
```

Python Mapping for Local Classes

On this page:

Mapped Python Class

A local Slice class is mapped to a Python class with the same name. For example:

Slice

```
module M
{
    local class Example
    {
        ...
    }
}
```

is mapped to the Python class `Example`:

Python

```
class Example(object):
    ...
```

LocalObject in Python

Like local interfaces, local Slice classes implicitly derive from `LocalObject`, which is mapped to the native `object` class in Python.

Mapping for Local Interface Inheritance in Python

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. Both `extends` and `implements` are mapped to class inheritance in Python. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

Python

```
class A(object):
    ...
class B(object):
    ...
class C(A, B):
    ...
class D(C):
    ...
```

Python Mapping for Local Exceptions

On this page:

Mapped Python Class

A local Slice exception is mapped to a Python class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the Python class `InitializationException`:

Python

```
class InitializationException(LocalException):
    ...
```

Base Class for Local Exceptions in Python

All mapped Python classes for local exceptions extend the class `Ice.LocalException`:

Python

```
class LocalException(Exception):
    ...
```

`LocalException` derives from `Ice.Exception`, which derives from Python's native `Exception` class.

Mapping for Local Exception Inheritance in Python

A local Slice exception can extend another Slice exception, which is mapped to class inheritance in Python. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

Python

```
class ErrorBase(Ice.LocalException):
    ...
class ResourceError(ErrorBase):
    ...
```

Python Mapping for Operations on Local Types

An operation on a local interface or a local class is mapped to a Python method with the same name. The mapping of operation parameters to Python is identical to the client-side mapping for these parameters, in particular, when an operation has more than one return value and out parameters, the mapped Python method must return a tuple.

Unlike the client-side mapping, there is no mapped method with a trailing Context parameter.

For example:

Slice

```
module M
{
    local interface L; // forward declared
    local sequence<L> LSeq;
    local interface L
    {
        string op(int n, string s, LocalObject any, out int m, out
string t, out LSeq newLSeq);
    }
}
```

is mapped to:

Python

```
class L(object):
    def op(self, n, s, any):
        raise NotImplementedError("method 'op' not implemented")
```

Ruby Mapping

Ice currently provides a client-side mapping for Ruby, but not a server-side mapping.

Topics

Initialization in Ruby

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `Ice::Communicator` object.

A `Communicator` is a local Ruby object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application.

You initialize the Ice run time by calling `Ice::initialize`, for example:

Ruby

```
require 'Ice'

communicator = Ice::initialize(ARGV)
```

`Ice::initialize` accepts the argument list that is passed to the program by the operating system. The function scans the argument list for any command-line options that are relevant to the Ice run time; any such options are removed from the argument list so, when `Ice::initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your program, you must call `Communicator.destroy`. The `destroy` method is responsible for finalizing the Ice run time. In particular, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your program to terminate without calling `destroy` first.

The general shape of our Ice Ruby application is therefore:

Ruby

```
require 'Ice'
begin
    communicator = Ice::initialize(ARGV)
    ...
ensure
    if defined? communicator and communicator != nil
        communicator.destroy()
    end
end
```

This code is a little bit clunky, as we need to make sure the communicator gets destroyed in all paths, including when an exception is thrown.

Fortunately, the `Ice::initialize` function accepts an optional block: if provided, `initialize` will create a communicator, pass it to the block, destroy the communicator automatically when the block completes, and return the block's result as the result of `initialize`.

The preferred way to initialize the Ice run time in Ruby is therefore:

Ruby

```
require 'Ice'
Ice::initialize(ARGV) do |communicator, args|
    ...
end
```

`initialize` requires the block to accept one or two arguments: if the block accepts only one argument, `initialize` passes the communicator, otherwise `initialize` passes the communicator and the filtered argument vector.

See Also

- Communicator
- Communicator Initialization
- Communicator Shutdown and Destruction

Client-Side Slice-to-Ruby Mapping

The client-side Slice-to-Ruby mapping defines how Slice data types are translated to Ruby types, and how clients invoke operations, pass parameters, and handle errors. Much of the Ruby mapping is intuitive. For example, Slice sequences map to Ruby arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in Ruby.

The Ruby API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data — the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least the mappings for exceptions, interfaces, and operations in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

> In order to use the Ruby mapping, you should need no more than the Slice definition of your application and knowledge of the Ruby mapping rules. In particular, looking through the generated code in order to discern how to use the Ruby mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

> **The `Ice` Module**
> All of the APIs for the Ice run time are nested in the `Ice` module, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` module are generated from Slice definitions; other parts of the `Ice` module provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` m odule throughout the remainder of the manual.
>
> A Ruby application can load the Ice run time using the `require` statement:
>
> ```
> require 'Ice'
> ```
>
> If the statement executes without error, the Ice run time is loaded and available for use. You can determine the version of the Ice run time you have just loaded by calling the `stringVersion` function:
>
> ```
> icever = Ice::stringVersion()
> ```

Topics

Ruby Mapping for Identifiers

A Slice identifier maps to an identical Ruby identifier. For example, the Slice identifier `Clock` becomes the Ruby identifier `Clock`. There are two exceptions to this rule:

1. Ruby requires the names of classes, modules, and constants to begin with an upper case letter. If a Slice identifier maps to the name of a Ruby class, module, or constant, and the Slice identifier does not begin with an upper case letter, the mapping replaces the leading character with its upper case equivalent. For example, the Slice identifier `bankAccount` is mapped as `BankAccount`.
2. If a Slice identifier is the same as a Ruby keyword, the corresponding Ruby identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.

> You should try to avoid such identifiers as much as possible.

See Also

- Lexical Rules
- Ruby Mapping for Modules
- Ruby Mapping for Built-In Types
- Ruby Mapping for Enumerations
- Ruby Mapping for Structures
- Ruby Mapping for Sequences
- Ruby Mapping for Dictionaries
- Ruby Mapping for Constants
- Ruby Mapping for Exceptions
- Ruby Mapping for Interfaces
- Ruby Mapping for Operations

Ruby Mapping for Modules

A Slice module maps to a Ruby module with the same name. The mapping preserves the nesting of the Slice definitions.

See Also

- Modules
- Ruby Mapping for Identifiers
- Ruby Mapping for Built-In Types
- Ruby Mapping for Enumerations
- Ruby Mapping for Structures
- Ruby Mapping for Sequences
- Ruby Mapping for Dictionaries
- Ruby Mapping for Constants
- Ruby Mapping for Exceptions
- Ruby Mapping for Interfaces
- Ruby Mapping for Operations

Ruby Mapping for Built-In Types

On this page:

Mapping of Slice Built-In Types to Ruby Types

The Slice built-in types are mapped to Ruby types as shown in this table:

Slice	Ruby
`bool`	`true` or `false`
`byte`	`Fixnum`
`short`	`Fixnum`
`int`	`Fixnum` or `Bignum`
`long`	`Fixnum` or `Bignum`
`float`	`Float`
`double`	`Float`
`string`	`String`

Although Ruby supports arbitrary precision in its integer types, the Ice run time validates integer values to ensure they have valid ranges for their declared Slice types.

String Mapping in Ruby 1.8

String values returned as the result of a Slice operation (including return values, out parameters, and data members) contain UTF-8 encoded strings unless the program has installed a string converter, in which case string values use the converter's native encoding instead.

As string input values for a remote Slice operation, Ice accepts `nil` in addition to `String` objects; each occurrence of `nil` is marshaled as an empty string. Ice assumes that all `String` objects contain valid UTF-8 encoded strings unless the program has installed a string converter, in which case Ice assumes that `String` objects use the native encoding expected by the converter.

String Mapping in Ruby 1.9 and later

String values returned as the result of a Slice operation (including return values, out parameters, and data members) contain UTF-8 encoded strings.

As string input values for a remote Slice operation, Ice accepts `nil` in addition to `String` objects; each occurrence of `nil` is marshaled as an empty string. Ice assumes that all `String` objects contain valid UTF-8 encoded strings.

The string converter facility is not used.

See Also

- Basic Types
- Ruby Mapping for Identifiers
- Ruby Mapping for Modules
- Ruby Mapping for Enumerations
- Ruby Mapping for Structures
- Ruby Mapping for Sequences
- Ruby Mapping for Dictionaries
- Ruby Mapping for Constants
- Ruby Mapping for Exceptions
- Ruby Mapping for Interfaces

Ruby Mapping for Enumerations

Ruby does not have an enumerated type, so a Slice enumeration is emulated using a Ruby class: the name of the Slice enumeration becomes the name of the Ruby class; for each enumerator, the class contains a constant with the same name as the enumerator. For example:

Slice

```
enum Fruit { Apple, Pear, Orange }
```

The generated Ruby class looks as follows:

Ruby

```
class Fruit
    include Comparable

    Apple = # ...
    Pear = # ...
    Orange = # ...

    def Fruit.from_int(val)

    def to_i

    def to_s

    def <=>(other)

    def hash

    # ...
end
```

The compiler generates a class constant for each enumerator that holds a corresponding instance of `Fruit`. The `from_int` class method returns an instance given its Slice value, while `to_i` returns the Slice value of an enumerator and `to_s` returns its Slice identifier.

Given the above definitions, we can use enumerated values as follows:

Ruby

```
f1 = Fruit::Apple
f2 = Fruit::Orange

if f1 == Fruit::Apple   # Compare for equality
    # ...

if f1 < f2              # Compare two enums
    # ...

case f2
when Fruit::Orange
    puts "found Orange"
else
    puts "found #{f2.to_s}"
end
```

Comparison operators are available as a result of including `Comparable`, which means a program can compare enumerators according to their Slice values. Note that, when using custom enumerator values, the order of enumerators by their Slice values may not match their order of declaration.

Suppose we modify the Slice definition to include a custom enumerator value:

Slice

```
enum Fruit { Apple, Pear = 3, Orange }
```

We can use `from_int` to examine the Slice values of the enumerators:

Ruby

```
Fruit::from_int(0)  # Apple
Fruit::from_int(1)  # nil
Fruit::from_int(3)  # Pear
Fruit::from_int(4)  # Orange
```

See Also

- Enumerations
- Ruby Mapping for Identifiers
- Ruby Mapping for Modules
- Ruby Mapping for Built-In Types
- Ruby Mapping for Structures
- Ruby Mapping for Sequences
- Ruby Mapping for Dictionaries
- Ruby Mapping for Constants
- Ruby Mapping for Exceptions
- Ruby Mapping for Interfaces
- Ruby Mapping for Operations

Ruby Mapping for Structures

A Slice structure maps to a Ruby class with the same name. For each Slice data member, the Ruby class contains a corresponding instance variable as well as accessors to read and write its value. For example, here is our `Employee` structure once more:

Slice

```
struct Employee
{
    long number;
    string firstName;
    string lastName;
}
```

The Ruby mapping generates the following definition for this structure:

Ruby

```
class Employee
    def initialize(number=0, firstName='', lastName='')
        @number = number
        @firstName = firstName
        @lastName = lastName
    end

    def hash
        # ...
    end

    def ==
        # ...
    end

    def inspect
        # ...
    end

    attr_accessor :number, :firstName, :lastName
end
```

The constructor initializes each of the instance variables to a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero

`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

You can also declare different default values for members of primitive and enumerated types.

The compiler generates a definition for the `hash` method, which allows instances to be used as keys in a hash collection. The `hash` method returns a hash value for the structure based on the value of its data members.

The == method returns true if all members of two structures are (recursively) equal.

The `inspect` method returns a string representation of the structure.

See Also

- Structures
- Ruby Mapping for Identifiers
- Ruby Mapping for Modules
- Ruby Mapping for Built-In Types
- Ruby Mapping for Enumerations
- Ruby Mapping for Sequences
- Ruby Mapping for Dictionaries
- Ruby Mapping for Constants
- Ruby Mapping for Exceptions
- Ruby Mapping for Interfaces
- Ruby Mapping for Operations

Ruby Mapping for Sequences

On this page:

Mapping Slice Sequences to Ruby Arrays

A Slice sequence maps to a Ruby array; the only exception is a sequence of bytes, which maps to a string. The use of a Ruby array means that the mapping does not generate a separate named type for a Slice sequence. It also means that you can take advantage of all the array functionality provided by Ruby. For example:

Slice

```
sequence<Fruit> FruitPlatter;
```

We can use the `FruitPlatter` sequence as shown below:

Ruby

```
platter = [ Fruit::Apple, Fruit::Pear ]
platter.push(Fruit::Orange)
```

The Ice run time validates the elements of a sequence to ensure that they are compatible with the declared type; a `TypeError` exception is raised if an incompatible type is encountered.

Mapping for Byte Sequences in Ruby

A Ruby string can contain arbitrary 8-bit binary data, therefore it is a more efficient representation of a byte sequence than a Ruby array in both memory utilization and throughput performance.

When receiving a byte sequence (as the result of an operation, as an out parameter, or as a member of a data structure), the value is always represented as a string. When sending a byte sequence as an operation parameter or data member, the Ice run time accepts both a string and an array of integers as legal values. For example, consider the following Slice definitions:

Slice

```
// Slice
sequence<byte> Data;

interface I
{
    void sendData(Data d);
    Data getData();
}
```

The interpreter session below uses these Slice definitions to demonstrate the mapping for a sequence of bytes:

Ruby

```
> proxy = ...
> proxy.sendData("\0\1\2\3")   # Send as a string
> proxy.sendData([0, 1, 2, 3]) # Send as an array
> d = proxy.getData()
> d.class
=> String
> d
=> "\000\001\002\003"
```

The two invocations of `sendData` are equivalent; however, the second invocation incurs additional overhead as the Ice run time must validate the type and range of each array element.

See Also

- Sequences
- Ruby Mapping for Identifiers
- Ruby Mapping for Modules
- Ruby Mapping for Built-In Types
- Ruby Mapping for Enumerations
- Ruby Mapping for Structures
- Ruby Mapping for Dictionaries
- Ruby Mapping for Constants
- Ruby Mapping for Exceptions
- Ruby Mapping for Interfaces
- Ruby Mapping for Operations

Ruby Mapping for Dictionaries

Here is the definition of our `EmployeeMap` once more:

Slice

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Ruby mapping does not create a separate named type for this definition. Instead, *all* dictionaries are simply instances of Ruby's hash collection type. For example:

Ruby

```
em = {}

e = Employee.new
e.number = 31
e.firstName = "James"
e.lastName = "Gosling"

em[e.number] = e
```

The Ice run time validates the elements of a dictionary to ensure that they are compatible with the declared type; a `TypeError` exception is raised if an incompatible type is encountered.

See Also

- Dictionaries
- Ruby Mapping for Identifiers
- Ruby Mapping for Modules
- Ruby Mapping for Built-In Types
- Ruby Mapping for Enumerations
- Ruby Mapping for Structures
- Ruby Mapping for Sequences
- Ruby Mapping for Constants
- Ruby Mapping for Exceptions
- Ruby Mapping for Interfaces
- Ruby Mapping for Operations

Ruby Mapping for Constants

Here are the constant definitions once more:

Slice

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange }
const Fruit     FavoriteFruit = Pear;
```

The generated definitions for these constants are shown below:

Ruby

```
AppendByDefault = true
LowerNibble = 15
Advice = "Don't Panic!"
TheAnswer = 42
PI = 3.1416
FavoriteFruit = Fruit::Pear
```

As you can see, each Slice constant is mapped to a Ruby constant with the same name.

Slice string literals that contain non-ASCII characters or universal character names are mapped to Ruby string literals with these characters replaced by their UTF-8 encoding as octal escapes. For example:

Slice

```
const string Egg = "œuf";
const string Heart = "c\u0153ur";
const string Banana = "\U0001F34C";
```

is mapped to:

Ruby

```
Egg = "\305\223uf"
Heart = "c\305\223ur"
Banana = "\360\237\215\214"
```

See Also

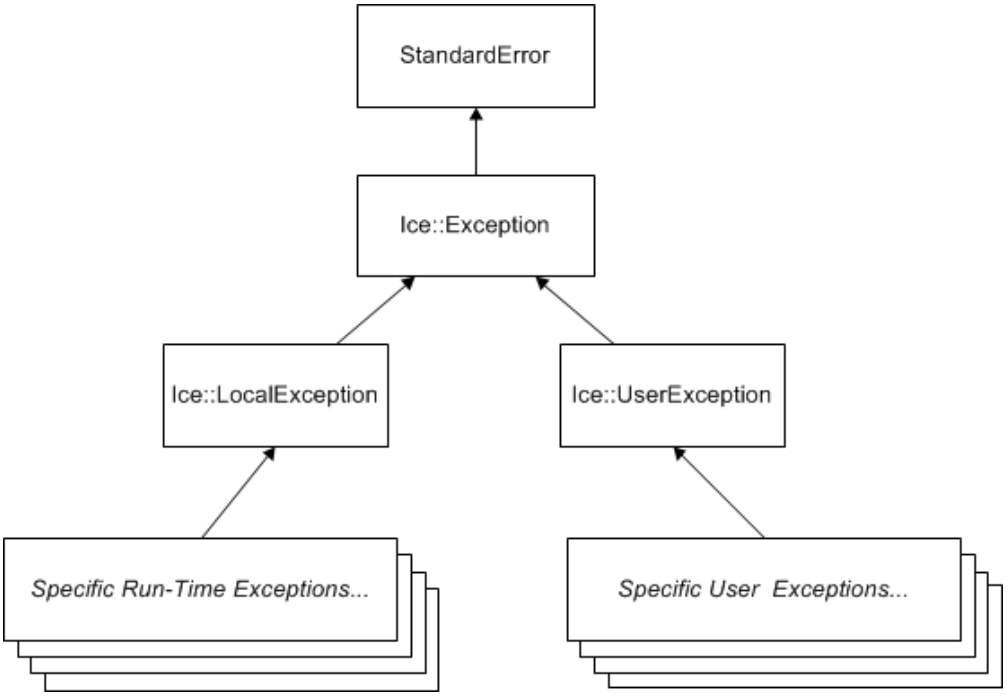
- Constants and Literals
- Ruby Mapping for Identifiers
- Ruby Mapping for Modules
- Ruby Mapping for Built-In Types

Ruby Mapping for Exceptions

On this page:

Inheritance Hierarchy for Exceptions in Ruby

The mapping for exceptions is based on the inheritance hierarchy shown below:



Inheritance structure for Ice exceptions.

The ancestor of all exceptions is `StandardError`, from which `Ice::Exception` is derived. `Ice::LocalException` and `Ice::UserException` are derived from `Ice::Exception` and form the base for all run-time and user exceptions.

Ruby Mapping for User Exceptions

Here is a fragment of the Slice definition for our world time server once more:

Slice

```
exception GenericError
{
    string reason;
}
exception BadTimeVal extends GenericError {}
exception BadZoneName extends GenericError {}
```

These exception definitions map to the abbreviated Ruby class definitions shown below:

Ruby

```
class GenericError < Ice::UserException
    def initialize(reason='')

    def to_s

    def inspect

    attr_accessor :reason
end

class BadTimeVal < GenericError
    def initialize(reason='')

    def to_s

    def inspect
end

class BadZoneName < GenericError
    def initialize(reason='')

    def to_s

    def inspect
end
```

Each Slice exception is mapped to a Ruby class with the same name. The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception member corresponds to an instance variable of the instance, which the constructor initializes to a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

You can also declare different default values for members of primitive and enumerated types. For derived exceptions, the constructor has one parameter for each of the base exception's data members, plus one parameter for each of the derived exception's data members, in base-to-derived order. As an example, although `BadTimeVal` and `BadZoneName` do not declare data members, their constructors still

accept a value for the inherited data member `reason` in order to pass it to the constructor of the base exception `GenericError`. The generated class defines an accessor for each data member to read and write its value.

Each exception also defines the standard methods `to_s` and `inspect` that return the Slice type ID of the exception and a stringified representation of the exception and its members, respectively. The method `ice_id` returns the same as `to_s`.

All user exceptions are derived from the base class `Ice::UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice::UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice::LocalException`, and you can catch all Ice exceptions with a handler for `Ice::Exception`.

Optional Data Members

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice::Unset` to indicate that the member is unset. A well-behaved program must compare an optional data member to `Ice::Unset` before using the member's value:

Ruby

```
begin
    ...
rescue => ex
    if ex.optionalMember == Ice::Unset
        puts "optionalMember is unset"
    else
        puts "optionalMember = " + ex.optionalMember
    end
end
```

The `Ice::Unset` marker value has different semantics than `nil`. Since `nil` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to `nil` is considered to be set. If you need to distinguish between an unset value and a value set to `nil`, you can do so as follows:

Ruby

```
begin
    ...
rescue => ex
    if ex.optionalMember == Ice::Unset
        puts "optionalMember is unset"
    elsif ex.optionalMember == nil
        puts "optionalMember is nil"
    else
        puts "optionalMember = " + ex.optionalMember
    end
end
```

Ruby Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice::LocalException` (which, in turn, derives from `Ice::Exception`).

By catching exceptions at the appropriate point in the inheritance hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice::LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `Ice::UserException`
  This is the root of the inheritance tree for user exceptions.
- `Ice::TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `Ice::ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `FacetNotExistException` and `ObjectNotExistException`, respectively.

See Also

- User Exceptions
- Run-Time Exceptions
- Ruby Mapping for Identifiers
- Ruby Mapping for Modules
- Ruby Mapping for Built-In Types
- Ruby Mapping for Enumerations
- Ruby Mapping for Structures
- Ruby Mapping for Sequences
- Ruby Mapping for Dictionaries
- Ruby Mapping for Constants
- Optional Data Members
- Versioning
- Object Life Cycle

Ruby Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Classes in Ruby

On the client side, a Slice interface maps to a Ruby class with methods that correspond to the operations on those interfaces. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Ruby mapping generates the following definition for use by the client:

Ruby

```
class SimplePrx < Ice::ObjectPrx
    def op(context=nil)
        # ...
    end

    def SimplePrx.ice_staticId()
        # ...
    end
    # ...
end
```

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice::ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. In the preceding example, we find that the operation `op` has been mapped to the method `op`. Note that `op` accepts an optional trailing parameter `context` representing the operation context. This parameter is a Ruby hash value for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the context parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of `nil` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Another method defined by every proxy class is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Interface Inheritance in Ruby

Inheritance relationships among Slice interfaces are maintained in the generated Ruby classes. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

The generated code for `CPrx` uses mixins to include the operations of `APrx` and `BPrx`:

Ruby

```
module CPrx_mixin
    include APrx_mixin
    include BPrx_mixin
end

class CPrx < ::Ice::ObjectPrx
    include ::Ice::Proxy_mixin
    include CPrx_mixin
end
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

`Ice::ObjectPrx` Class in Ruby

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice::ObjectPrx`. `ObjectPrx` provides a number of methods:

Ruby

```
class ObjectPrx
    def eql?(proxy)
    def ice_getIdentity
    def ice_isA(id)
    def ice_ids
    def ice_id
    def ice_ping
    # ...
end
```

The methods behave as follows:

- **`eql?`**
  The implementation of this standard Ruby method compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `eql?` returns `false` even though the proxies denote the same object.

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

  ```
  module Ice
  {
      struct Identity
      {
          string name;
          string category;
      }
  }
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

  Ruby

  ```
  proxy1 = ...
  proxy2 = ...
  id1 = proxy1.ice_getIdentity
  id2 = proxy2.ice_getIdentity

  if id1 == id2
      # proxy1 and proxy2 denote the same object
  else
      # proxy1 and proxy2 denote different objects
  end
  ```

- **`ice_isA`**
  The `ice_isA` method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID. For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

  Ruby

  ```
  proxy = ...
  if proxy && proxy.ice_isA("::Printer")
      # proxy denotes a Printer object
  else
      # proxy denotes some other type of object
  end
  ```

  Note that we are testing whether the proxy is `nil` before attempting to invoke the `ice_isA` method. This avoids getting a run-time error if the proxy is `nil`.

- **`ice_ids`**

The `ice_ids` method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- **`ice_id`**
  The `ice_id` method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the return value of `ice_id` might be `"::Base"`, or it might be something more derived, such as `"::Derived"`.

- **`ice_ping`**
  The `ice_ping` method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExistException` or `ConnectTimeoutException`.

The `ice_isA`, `ice_ids`, `ice_id`, and `ice_ping` methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Casting Proxies in Ruby

The Ruby mapping for a proxy also generates two class methods:

Ruby

```
class SimplePrx < Ice::ObjectPrx
    # ...

    def SimplePrx.checkedCast(proxy, facet='', context={})

    def SimplePrx.uncheckedCast(proxy, facet='')
end
```

The method names `checkedCast` and `uncheckedCast` are reserved for use in proxies. If a Slice interface defines an operation with either of those names, the mapping escapes the name in the generated proxy by prepending an underscore. For example, an interface that defines an operation named `checkedCast` is mapped to a proxy with a method named `_checkedCast`.

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is `nil`), the cast returns `nil`.

Given a proxy of any type, you can use a checkedCast to determine whether the corresponding object supports a given type, for example:

Ruby

```
obj = ...       # Get a proxy from somewhere...

simple = SimplePrx::checkedCast(obj)
if simple
    # Object supports the Simple interface...
else
    # Object is not of type Simple...
end
```

Note that a `checkedCast` contacts the server. This is necessary because only the server implementation has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong,

you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

Ruby

```
obj = ...                                # Get proxy...
process = ProcessPrx::uncheckedCast(obj) # No worries...
process.launch(40, 60)                   # Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

Using Proxy Methods in Ruby

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

Ruby

```
proxy = communicator.stringToProxy(...)
proxy = proxy.ice_invocationTimeout(10000)
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

Ruby

```
base = communicator.stringToProxy(...)
hello = Demo::HelloPrx::checkedCast(base)
hello = hello.ice_invocationTimeout(10000) # Type is not discarded
hello.sayHello()
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in Ruby

Proxy objects support comparison using the comparison operators ==, `!=`, and <=>, as well as the `eql?` method. Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

Ruby

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if p1 != p2
    # p1 and p2 denote different objects       # WRONG!
else
    # p1 and p2 denote the same object         # Correct
end
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each uses a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` module:

Ruby

```
def proxyIdentityCompare(lhs, rhs)
def proxyIdentityAndFacetCompare(lhs, rhs)
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

Ruby

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if Ice.proxyIdentityCompare(p1, p2) != 0
    # p1 and p2 denote different objects       # Correct
else
    # p1 and p2 denote the same object         # Correct
end
```

The function returns `0` if the identities are equal, `-1` if `p1` is less than `p2`, and `1` if `p1` is greater than `p2`. (The comparison uses `name` as the major sort key and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Type IDs
- Ruby Mapping for Operations
- Request Contexts
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

Ruby Mapping for Operations

On this page:

Basic Ruby Mapping for Operations

As we saw in the Ruby mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding method with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

Ruby

```
node = ...          # Initialize proxy
name = node.name()  # Get name via RPC
```

Normal and `idempotent` Operations in Ruby

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, idempotent has no effect. For example, consider the following interface:

Slice

```
interface Example
{
                string op1();
    idempotent  string op2();
}
```

The proxy class for this is:

Ruby

```
class ExamplePrx < Ice::ObjectPrx
    def op1(context=nil)

    def op2(context=nil)
end
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to look the same.

Passing Parameters in Ruby

In-Parameters in Ruby

All parameters are passed by reference in the Ruby mapping; it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
}
```

The Slice compiler generates the following proxy for this definition:

Ruby

```
class ClientToServerPrx < Ice::ObjectPrx
    def op1(i, f, b, s, context=nil)

    def op2(ns, ss, st, context=nil)

    def op3(proxy, context=nil)
end
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

Ruby

```
p = ...                                 # Get proxy...

p.op1(42, 3.14, true, "Hello world!")   # Pass simple literals

i = 42
f = 3.14
b = true
s = "Hello world!"
p.op1(i, f, b, s)                       # Pass simple variables

ns = NumberAndString.new()
ns.x = 42
ns.str = "The Answer"
ss = [ "Hello world!" ]
st = {}
st[0] = ns
p.op2(ns, ss, st)                       # Pass complex variables

p.op3(p)                                # Pass proxy
```

Out-Parameters in Ruby

As in Java, Ruby functions do not support reference arguments. That is, it is not possible to pass an uninitialized variable to a Ruby function in order to have its value initialized by the function. The Java mapping overcomes this limitation with the use of *holder classes* that represent each `out` parameter. The Ruby mapping takes a different approach, one that is more natural for Ruby users.

The semantics of `out` parameters in the Ruby mapping depend on whether the operation returns one value or multiple values. An operation returns multiple values when it has declared multiple `out` parameters, or when it has declared a non-`void` return type and at least one `out` parameter.

If an operation returns multiple values, the client receives them in the form of a *result array*. A non-`void` return value, if any, is always the first element in the result array, followed by the `out` parameters in the order of declaration.

If an operation returns only one value, the client receives the value itself.

Here again are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString
{
    int x;
    string str;
}

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient
{
    int op1(out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
}
```

The Ruby mapping generates the following code for this definition:

Ruby

```
class ClientToServerPrx < Ice::ObjectPrx
    def op1(context=nil)

    def op2(context=nil)

    def op3(context=nil)
end
```

Given a proxy to a `ServerToClient` interface, the client code can receive the results as in the following example:

Ruby

```
p = ...              # Get proxy...
i, f, b, s = p.op1()
ns, ss, st = p.op2()
stcp = p.op3()
```

The operations have no `in` parameters, therefore no arguments are passed to the proxy methods. Since `op1` and `op2` return multiple values, their result arrays are unpacked into separate values, whereas the return value of `op3` requires no unpacking.

Parameter Type Mismatches in Ruby

Although the Ruby compiler cannot check the types of arguments passed to a method, the Ice run time does perform validation on the arguments to a proxy invocation and reports any type mismatches as a `TypeError` exception.

Null Parameters in Ruby

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `nil`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `nil` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, it makes no difference to the receiver whether you send a string as `nil` or as an empty string: either way, the receiver sees an empty string.

Optional Parameters in Ruby

Optional parameters use the same mapping as required parameters. The only difference is that `Ice::Unset` can be passed as the value of an optional parameter or return value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float
value);
```

A client can invoke this operation as shown below:

Ruby

```
i, v = proxy.execute("--file log.txt")
i, v = proxy.execute(Ice::Unset)

if v != Ice::Unset
    puts "value = " + v.to_s
end
```

A well-behaved program must always compare an optional parameter to `Ice::Unset` prior to using its value. Keep in mind that the `Ice::Unset` marker value has different semantics than `nil`. Since `nil` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional parameter is set. An optional parameter set to `nil` is considered to be set. If you need to distinguish between an unset parameter and a parameter set to `nil`, you can do so as follows:

Ruby

```
if optionalParam == Ice::Unset
    puts "optionalParam is unset"
elsif optionalParam == nil
    puts "optionalParam is None"
else
    puts "optionalParam = " + optionalParam
end
```

Exception Handling in Ruby

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum
{
    string reason;
}

interface Child
{
    void askToCleanUp() throws Tantrum;
}
```

Slice exceptions are thrown as Ruby exceptions, so you can simply enclose one or more operation invocations in a `begin-rescue` block:

Ruby

```
child = ...         # Get child proxy...

begin
    child.askToCleanUp()
rescue Tantrum => t
    puts "The child says: #{t.reason}"
end
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will usually be handled by exception handlers higher in the hierarchy. For example:

Ruby

```
begin
    child = ...             # Get child proxy...
    begin
        child.askToCleanUp()
        child.praise()      # Give positive feedback...
    rescue Tantrum => t
        puts "The child says: #{t.reason}"
        child.scold()       # Recover from error...
    end
rescue Ice::Exception => ex
    print ex.backtrace.join("\n")
end
```

See Also

- Operations
- Hello World Application
- Slice for a Simple File System
- Ruby Mapping for Interfaces

- Ruby Mapping for Exceptions

Ruby Mapping for Classes

On this page:

Basic Ruby Mapping for Classes

A Slice class maps to a Ruby class with the same name. For each Slice data member, the generated class contains an instance variable and accessors to read and write it, just as for structures and exceptions. Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

The Ruby mapping generates the following code for this definition:

Ruby

```
class TimeOfDay < ::Ice::Value
    def initialize(hour=0, minute=0, second=0)
        @hour = hour
        @minute = minute
        @second = second
    end

    attr_accessor :hours, :minutes, :seconds
end
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` derives from `Ice::Value`. This reflects the semantics of Slice classes in that all classes implicitly inherit from `Ice::Value`, which is the ultimate ancestor of all classes. Note that `Ice::Value` is *not* the same as `Ice::ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The constructor defines an instance variable for each Slice data member.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice::Value` in Ruby

Like interfaces, classes implicitly inherit from a common base class, `Ice::Value`. However classes inherit from `Ice::Value` instead of `Ice::ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`Ice::Value` contains a number of methods:

Ruby

```
class Value
    def inspect
        ::Ice::__stringify(self, self.class::ICE_TYPE)
    end
    def ice_id()
        self.class::ICE_ID
    end
    def Value.ice_staticId()
        self::ICE_ID
    end
end
```

The methods behave as follows:

- `ice_id`
  This method returns the actual run-time type ID of the object. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_staticId`
  This method returns the static type ID of the class.
- `ice_preMarshal`
  If the object supports this method, the Ice run time invokes it just prior to marshaling the object's state, providing the opportunity for the object to validate its declared data members.
- `ice_postUnmarshal`
  If the object supports this method, the Ice run time invokes it after unmarshaling the object's state. An object typically defines this method when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `nil` otherwise.

Note that neither `Ice::Value` nor the generated class override `hash` and ==, so the default implementations apply.

Class Data Members in Ruby

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding instance variable and accessor methods.

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice::Unset` to indicate that the member is unset. A well-behaved program must compare an optional data member to `Ice::Unset` before using the member's value:

Ruby

```
v = ...
if v.optionalMember == Ice::Unset
    puts "optionalMember is unset"
else
    puts "optionalMember = " + v.optionalMember
end
```

The `Ice::Unset` marker value has different semantics than `nil`. Since `nil` is a legal value for certain Slice types, the Ice run time requires

a separate marker value so that it can determine whether an optional value is set. An optional value set to `nil` is considered to be set. If you need to distinguish between an unset value and a value set to `nil`, you can do so as follows:

Ruby

```
v = ...
if v.optionalMember == Ice::Unset
    puts "optionalMember is unset"
elsif v.optionalMember == nil
    puts "optionalMember is nil"
else
    puts "optionalMember = " + v.optionalMember
end
```

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
}
```

The Slice compiler produces the following generated code for this definition:

Ruby

```
class TimeOfDay < ::Ice::Value

    def initialize(hour=0, minute=0, second=0)
        @hour = hour
        @minute = minute
        @second = second
    end

    attr_accessor :hours, :minutes, :seconds
    protected :hours, :hours=
    protected :minutes, :minutes=
    protected :seconds, :seconds=
end
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

Class Constructors in Ruby

Classes have a constructor that assigns to each data member a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

You can also declare different default values for data members of primitive and enumerated types.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

Pass the marker value `Ice::Unset` as the value of any optional data members that you wish to be unset.

Class Operations in Ruby

Deprecated Feature
Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

With the Ruby mapping, operations in classes are not mapped at all into the corresponding Ruby class. The generated Ruby class is the same whether the Slice class has operations or not.

Value Factories in Ruby

While value factories were necessary in previous Ice versions when using classes with operations (a now deprecated feature) with the Ruby mapping, value factories may be used for any kind of class and are *not* deprecated.

Value factories allow you to create classes derived from the Ruby class generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
class CustomTimeOfDay extends TimeOfDay
{
    public function format() { ... prints formatted data members ... }
}
```

You then create and register a value factory for your custom class with your Ice communicator:

Ruby

```
class ValueFactory
    def create(type)
        fail unless type == M::TimeOfDay::ice_staticId()
        TimeOfDayI.new
    end
end

communicator = ...
communicator.getValueFactoryManager().add(ValueFactory.new,
M::TimeOfDay::ice_staticId())
```

See Also

- Classes
- Type IDs
- Optional Data Members
- The Current Object
- Value Factories

Code Generation in Ruby

The Ruby mapping supports two forms of code generation: dynamic and static.

On this page:

Dynamic Code Generation in Ruby

Using dynamic code generation, Slice files are "loaded" at run time and dynamically translated into Ruby code, which is immediately compiled and available for use by the application. This is accomplished using the `Ice::loadSlice` method, as shown in the following example:

Ruby

```
Ice::loadSlice("Color.ice")
puts "My favorite color is #{M::Color.blue.to_s}"
```

For this example, we assume that `Color.ice` contains the following definitions:

Slice

```
module M
{
    enum Color { red, green, blue }
}
```

`Ice::loadSlice` Options in Ruby

The `Ice::loadSlice` method behaves like a Slice compiler in that it accepts command-line arguments for specifying preprocessor options and controlling code generation. The arguments must include at least one Slice file.

The function has the following Ruby definition:

Ruby

```
def loadSlice(cmd, args=[])
```

The command-line arguments can be specified entirely in the first argument, `cmd`, which must be a string. The optional second argument can be used to pass additional command-line arguments as a list; this is useful when the caller already has the arguments in list form. The function always returns `nil`.

For example, the following calls to `Ice::loadSlice` are functionally equivalent:

Ruby

```
Ice::loadSlice("-I/opt/IceRuby/slice Color.ice")
Ice::loadSlice("-I/opt/IceRuby/slice", ["Color.ice"])
Ice::loadSlice("", ["-I/opt/IceRuby/slice", "Color.ice"])
```

In addition to the standard compiler options, `Ice::loadSlice` also supports the following command-line options:

- `--all`
  Generate code for all Slice definitions, including those from included files.
- `--checksum`
  Generate checksums for Slice definitions.

Locating Slice Files in Ruby

If your Slice files depend on Ice types, you can avoid hard-coding the path name of your Ice installation directory by calling the `Ice::getSliceDir` function:

Ruby

```
Ice::loadSlice("-I" + Ice::getSliceDir() + " Color.ice")
```

This function attempts to locate the `slice` subdirectory of your Ice installation using an algorithm that succeeds for the following scenarios:

- Installation of a binary Ice archive
- Installation of an Ice source distribution using `make install`
- Installation via a Windows installer
- RPM installation on Linux
- Execution inside a compiled Ice source distribution

If the `slice` subdirectory can be found, `getSliceDir` returns its absolute path name, otherwise the function returns `nil`.

Loading Multiple Slice Files in Ruby

You can specify as many Slice files as necessary in a single invocation of `Ice::loadSlice`, as shown below:

Ruby

```
Ice::loadSlice("Syscall.ice Process.ice")
```

Alternatively, you can call `Ice::loadSlice` several times:

Ruby

```
Ice::loadSlice("Syscall.ice")
Ice::loadSlice("Process.ice")
```

If a Slice file includes another file, the default behavior of `Ice::loadSlice` generates Ruby code only for the named file. For example, suppose `Syscall.ice` includes `Process.ice` as follows:

Slice

```
// Syscall.ice
#include <Process.ice>
...
```

If you call `Ice::loadSlice("-I. Syscall.ice")`, Ruby code is not generated for the Slice definitions in `Process.ice` or for any definitions that may be included by `Process.ice`. If you also need code to be generated for included files, one solution is to load them individually in subsequent calls to `Ice::loadSlice`. However, it is much simpler, not to mention more efficient, to use the `--all` option instead:

Ruby

```
Ice::loadSlice("--all -I. Syscall.ice")
```

When you specify `--all`, `Ice::loadSlice` generates Ruby code for all Slice definitions included directly or indirectly from the named Slice files.

There is no harm in loading a Slice file multiple times, aside from the additional overhead associated with code generation. For example, this situation could arise when you need to load multiple top-level Slice files that happen to include a common subset of nested files. Suppose that we need to load both `Syscall.ice` and `Kernel.ice`, both of which include `Process.ice`. The simplest way to load both files is with a single call to `Ice::loadSlice`:

Ruby

```
Ice::loadSlice("--all -I. Syscall.ice Kernel.ice")
```

Although this invocation causes the Ice extension to generate code twice for `Process.ice`, the generated code is structured so that the interpreter ignores duplicate definitions. We could have avoided generating unnecessary code with the following sequence of steps:

Ruby

```
Ice::loadSlice("--all -I. Syscall.ice")
Ice::loadSlice("-I. Kernel.ice")
```

In more complex cases, however, it can be difficult or impossible to completely avoid this situation, and the overhead of code generation is usually not significant enough to justify such an effort.

Limitations of Dynamic Code Generation in Ruby

The `Ice::loadSlice` method must be called outside of any module scope. For example, the following code is incorrect:

Ruby

```
# WRONG
module M
    Ice::loadSlice("--all -I. Syscall.ice Kernel.ice")
    ...
end
```

Static Code Generation in Ruby

You should be familiar with static code generation if you have used other Slice language mappings, such as C++ or Java. Using static code generation, the Slice compiler `slice2rb` generates Ruby code from your Slice definitions.

Compiler Output in Ruby

For each Slice file `X.ice`, `slice2rb` generates Ruby code into a file named `X.rb` in the output directory. The default output directory is the current working directory, but a different directory can be specified using the `--output-dir` option.

Include Files in Ruby

It is important to understand how `slice2rb` handles include files. In the absence of the `--all` option, the compiler does not generate Ruby code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into Ruby `require` statements in the following manner:

1. Determine the full pathname of the included file.
2. Create the shortest possible relative pathname for the included file by iterating over each of the include directories (specified using the `-I` option) and removing the leading directory from the included file if possible.
   For example, if the full pathname of an included file is `/opt/App/slice/OS/Process.ice`, and we specified the options `-I/opt/App` and `-I/opt/App/slice`, then the shortest relative pathname is `OS/Process.ice` after removing `/opt/App/slice`.
3. Replace the `.ice` extension with `.rb`. Continuing our example from the previous step, the translated `require` statement becomes

```
require "OS/Process.rb"
```

As a result, you can use `-I` options to tailor the `require` statements generated by the compiler in order to avoid absolute pathnames and match the organizational structure of your application's source files.

Static Versus Dynamic Code Generation in Ruby

There are several issues to consider when evaluating your requirements for code generation.

Application Considerations for Code Generation in Ruby

The requirements of your application generally dictate whether you should use dynamic or static code generation. Dynamic code generation is convenient for a number of reasons:

- It avoids the intermediate compilation step required by static code generation.
- It makes the application more compact because the application requires only the Slice files, not the additional files produced by static code generation.
- It reduces complexity, which is especially helpful during testing, or when writing short or transient programs.

Static code generation, on the other hand, is appropriate in many situations:

- when an application uses a large number of Slice definitions and the startup delay must be minimized
- when it is not feasible to deploy Slice files with the application
- when a number of applications share the same Slice files
- when Ruby code is required in order to utilize third-party Ruby tools.

Mixing Static and Dynamic Generation in Ruby

You can safely use a combination of static and dynamic translation in an application. For it to work properly, you must correctly manage the include paths for Slice translation and the Ruby interpreter so that the statically-generated code can be imported properly by `require`.

For example, suppose you want to dynamically load the following Slice definitions:

Slice

```
#include <Glacier2/Session.ice>

module MyApp
{
    interface MySession extends Glacier2::Session
    {
        // ...
    }
}
```

Whether the included file `Glacier2/Session.ice` is loaded dynamically or statically is determined by the presence of the `--all` option:

Ruby

```
sliceDir = "-I#{ENV['ICE_HOME']}/slice"

# Load Glacier2/Session.ice dynamically:
Ice::loadSlice(sliceDir + " --all MySession.ice")

# Load Glacier2/Session.ice statically:
Ice::loadSlice(sliceDir + " MySession.ice")
```

In this example, the first invocation of `loadSlice` uses the `--all` option so that code is generated dynamically for all included files. The second invocation omits `--all`, therefore the Ruby interpreter executes the equivalent of the following statement:

```
require "Glacier2/Session.rb"
```

As a result, before we can call `loadSlice` we must first ensure that the interpreter can locate the statically-generated file `Glacier2/Session.rb`. We can do this in a number of ways, including:

- adding the parent directory (e.g., `/opt/IceRuby/ruby`) to the `RUBYLIB` environment variable
- specifying the `-I` option when starting the interpreter
- modifying the search path at run time, as shown below:

```
$:.unshift("/opt/IceRuby/ruby")
```

`slice2rb` Command-Line Options

The Slice-to-Ruby compiler, `slice2rb`, offers the following command-line options in addition to the standard options:

- `--all`
  Generate code for all Slice definitions, including those from included files.
- `--checksum`
  Generate checksums for Slice definitions.

See Also

- Using the Slice Compilers
- Using Slice Checksums in Ruby

Using Slice Checksums in Ruby

The Slice compilers can optionally generate checksums of Slice definitions. For `slice2rb`, the `--checksum` option causes the compiler to generate code that adds checksums to the hash collection `Ice::SliceChecksums`. The checksums are installed automatically when the Ruby code is first parsed; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the two hash objects using a comparison operator. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

Ruby

```
serverChecksums = ...
for i in Ice::SliceChecksums.keys
    if not serverChecksums.has_key?(i)
        # No match found for type id!
    elsif Ice::SliceChecksums[i] != serverChecksums[i]
        # Checksum mismatch!
    end
end
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

See Also

- Slice Checksums

Example of a File System Client in Ruby

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The Ruby code shown here hardly differs from the code you would write for an ordinary Ruby program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Ruby object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs.

We now have seen enough of the client-side Ruby mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }

    exception GenericError
    {
        string reason;
    }

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    sequence<Node*> NodeSeq;

    interface Directory extends Node
    {
        idempotent NodeSeq list();
    }
}
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

Ruby

```
require 'Filesystem.rb'

# Recursively print the contents of directory "dir"
# in tree fashion. For files, show the contents of
# each file. The "depth" parameter is the current
# nesting level (for indentation).

def listRecursive(dir, depth)
    indent = ''
    depth = depth + 1
    for i in (0...depth)
        indent += "\t"
    end

    contents = dir.list()

    for node in contents
        subdir = Filesystem::DirectoryPrx::checkedCast(node)
        file = Filesystem::FilePrx::uncheckedCast(node)
        print indent + node.name()
        if subdir
            puts "(directory):"
            listRecursive(subdir, depth)
        else
            puts "(file):"
            text = file.read()
            for line in text
                puts indent + "\t" + line
            end
        end
    end
end

Ice::initialize(ARGV) do |communicator|
    # Create a proxy for the root directory
    #
    obj = communicator.stringToProxy("RootDir:default -h localhost -p
10000")

    # Down-cast the proxy to a Directory proxy
    #
    rootDir = Filesystem::DirectoryPrx::checkedCast(obj)

    # Recursively list the contents of the root directory
    #
    puts "Contents of root directory:"
    listRecursive(rootDir, 0)
end
```

The program first defines the `listRecursive` function, which is a helper function to print the contents of the file system, and the main

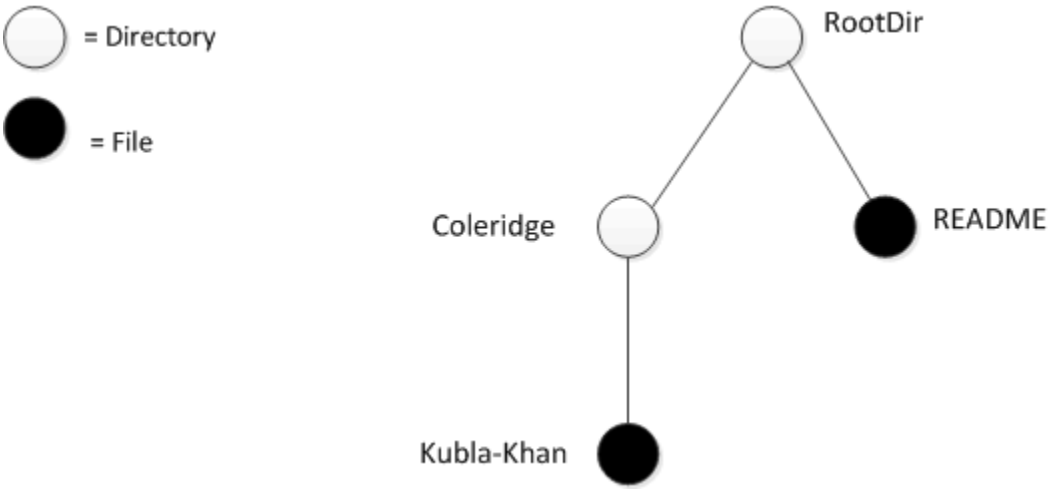
program follows. Let us look at the main program first:

1. The structure of the code follows what we saw in Hello World Application. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of a two files and a a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client is not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Object Life Cycle
- IceGrid

Slice-to-Ruby Mapping for Local Types

The mapping for `local enum`, `local sequence`, `local dictionary` and `local struct` to Ruby is identical to the mapping for these constructs without the `local` qualifier. The generated Ruby code for local enums and structs does not include support for marshaling, so you cannot use them as parameters for operations on non-local types, or as data members on non-local types.

Data members on local Slice types (classes, exceptions and structs) are mapped to Ruby just like the data members of the corresponding non-local Slice construct.

A `local interface` has no mapping in Ruby since the language is loosely typed and has no equivalent for an interface.

An operation defined in a `local class` or `local interface` generates no code in Ruby. A method implementation has the same signature as the client-side mapping without the trailing `Context` parameter.

The rest of this section describes the mapping of the remaining local types to Ruby:

- Ruby Mapping for Local Classes
- Ruby Mapping for Local Exceptions

Ruby Mapping for Local Classes

On this page:

Mapped Ruby Class

A local Slice class is mapped to a Ruby class with the same name. For example:

Slice

```
module M
{
    local class Example
    {
        ...
    }
}
```

is mapped to the Ruby class `Example`:

Ruby

```
class Example
    ...
end
```

LocalObject in Ruby

Local Slice classes implicitly derive from `LocalObject`, which is mapped to the native `Object` class in Ruby.

Mapping for Local Interface Inheritance in Ruby

A local Slice class can extend another local Slice class, and can implement one or more local Slice interfaces. `extends` is mapped to class inheritance in Ruby, but `implements` is not mapped. For example:

Slice

```
module M
{
    local interface A {}
    local interface B {}

    local class C implements A, B {}
    local class D extends C {}
}
```

is mapped to:

Ruby

```
# No mapping for A or B
class C
    ...
end
class D < C
    ...
end
```

Ruby Mapping for Local Exceptions

On this page:

Mapped Ruby Class

A local Slice exception is mapped to a Ruby class with the same name. For example:

Slice

```
module Ice
{
    local exception InitializationException
    {
        ...
    }
}
```

is mapped to the Ruby class `InitializationException`:

Ruby

```
class InitializationException < Ice::LocalException
    ...
end
```

Base Class for Local Exceptions in Ruby

All mapped Ruby classes for local exceptions extend the class `Ice::LocalException`:

Ruby

```
class LocalException < Exception
    ...
end
```

`LocalException` derives from `Ice::Exception`, which derives from Ruby's native `StandardError` class.

Mapping for Local Exception Inheritance in Ruby

A local Slice exception can extend another Slice exception, which is mapped to class inheritance in Ruby. For example:

Slice

```
module M
{
    local exception ErrorBase {}
    local exception ResourceError extends ErrorBase {}
}
```

is mapped to:

Ruby

```
class ErrorBase < Ice::LocalException
    ...
end
class ResourceError < ErrorBase
    ...
end
```

Properties and Configuration

Ice uses a configuration mechanism that allows you to control many aspects of the behavior of your Ice applications at run time, such as the maximum message size, the number of threads, or whether to produce network trace messages. The configuration mechanism is not only useful for configuring Ice, but also for configuring your own applications. The configuration mechanism is simple to use with a minimal API, yet flexible enough to cope with the needs of most applications.

Topics

Properties Overview

Ice and its various subsystems are configured by properties. A property is a name-value pair, for example:

```
Ice.UDP.SndSize=65535
```

In this example, the *property name* is `Ice.UDP.SndSize`, and the *property value* is `65535`.

You can find a complete list of the properties used to configure Ice in the property reference.

Note that Ice reads properties that control the Ice run time and its services (that is, properties that start with one of the reserved prefixes, such as `Ice`, `Glacier2`, etc.) only once on start-up, when you create a communicator. This means that you must set Ice-related properties to their correct values *before* you create a communicator. If you change the value of an Ice-related property after that point, it is likely that the new setting will simply be ignored.

On this page:

Property Categories

By convention, Ice properties use the following naming scheme:

```
<application>.<category>[.<sub-category>]
```

Note that the sub-category is optional and not used by all Ice properties.

This two- or three-part naming scheme is by convention only — if you use properties to configure your own applications, you can use property names with any number of categories.

Reserved Prefixes for Properties

Ice reserves properties with the following prefixes:

- `Ice`
- `IceBox`
- `IceBoxAdmin`
- `IceBT`
- `IceDiscovery`
- `IceGrid`
- `IceGridAdmin`
- `IceLocatorDiscovery`
- `IceMX`
- `IcePatch2`
- `IcePatch2Client`
- `IceSSL`
- `IceStorm`
- `IceStormAdmin`
- `Freeze`
- `Glacier2`

You cannot use a property beginning with one of these prefixes to configure your own application.

Property Name Syntax

A property name consists of any number of characters. For example, the following are valid property names:

```
foo
Foo
foo.bar
foo bar    White space is allowed
foo=bar    Special characters are allowed
.
```

Note that there is no special significance to a period in a property name. (Periods are used to make property names more readable and are not treated specially by the property parser.)

Property names cannot contain leading or trailing white space. (If you create a property name with leading or trailing white space, that white space is silently stripped.)

Property Value Syntax

A property value consists of any number of characters. The following are examples of property values:

```
65535
yes
This is a = property value.
../../config
```

Unused Properties

During the destruction of a communicator, the Ice run time can optionally emit a warning for properties that were set but never read. To enable this warning, set `Ice.Warn.UnusedProperties` to a non-zero value. This property is useful for detecting mis-spelled properties, such as `Filesystem.MaxFilSize`. By default, the warning is disabled.

See Also

- Property Reference

Configuration File Syntax

This page describes the syntax of an Ice configuration file.

On this page:

Configuration File Format

A configuration file contains any number of name-value pairs, with each pair on a separate line. Empty lines and lines consisting entirely of white space characters are ignored. The # character introduces a comment that extends to the end of the current line.

Configuration files can be ASCII text files or use the UTF-8 character encoding with an optional byte order marker (BOM) at the beginning of the file.

Here is a simple configuration file:

```
# Example config file for Ice

Ice.MessageSizeMax = 2048    # Largest message size is 2MB
Ice.Trace.Network=3          # Highest level of tracing for network
Ice.Trace.Protocol=          # Disable protocol tracing
```

White space within property keys and values is preserved, whether escaped with a backslash or not escaped.

Leading and trailing white space is always ignored for property *names* (whether the white space is escaped or not), for example:

```
# Key white space example

Prop1     = 1           # Key is "Prop1"
 Prop2    = 2           # Key is "Prop2"
\ Prop3 \ = 3           # Key is "Prop3"
My Prop1  = 1           # Key is "My Prop1"
My\ Prop2 = 2           # Key is "My Prop2"
```

For property values, you can preserve leading and trailing white space by escaping the white space with a backslash, for example:

```
# Value white space example

My.Prop1 = a property               # Value is "a property"
My.Prop2 =     a     property       # Value is "a     property"
My.Prop3 = \ \ a     property\ \    # Value is "  a     property  "
My.Prop4 = \ \ a  \ \  property\ \  # Value is "  a     property  "
My.Prop5 = a \\ property            # Value is "a \ property"
```

This example shows that leading and trailing white space for property values is ignored unless escaped with a backslash whereas, white space that is surrounded by non-white space characters is preserved exactly, whether it is escaped or not. As usual, you can insert a literal backslash into a property value by using a double backslash.

If you set the same property more than once, the last setting prevails and overrides any previous setting. Note that assigning nothing to a property clears that property (that is, sets it to the empty string).

Ice treats properties that contain the empty string (such as `Ice.Trace.Protocol` in the preceding example) like a property that is not set at all, and we recommend that your Ice-based applications do the same. With `getProperty`, `getPropertyAsInt`, `getPropertyAsIntWithDefault`, `getPropertyAsList` and `getPropertyAsListWithDefault`, you cannot distinguish between a property that is not set and a property set to the empty string; however, `getPropertyWithDefault` allows you to make this distinction, for example:

C++

```
// returns 3 if not set or set to the empty string
int traceProtocol =
properties->getPropertyAsIntWithDefault("Ice.Trace.Protocol", 3);

// returns "3" if not set but "" if set to the empty string
string traceProtocolString =
properties->getPropertyWithDefault("Ice.Trace.Protocol", "3");
```

Property values can include characters from non-English alphabets. The Ice run time expects the configuration file to use UTF-8 encoding for such characters. (With C++, you can specify a string converter when you read the file.)

Special Characters in Configuration Files

The characters = and # have special meaning in a configuration file:

- = marks the end of the property name and the beginning of the property value
- # starts a comment that extends to the end of the line

These characters must be escaped when they appear in a property name. Consider the following examples:

```
foo\=bar=1        # Name is "foo=bar", value is "1"
foo\#bar   = 2    # Name is "foo#bar", value is "2"
foo bar  =3       # Name is "foo bar", value is "3"
```

In a property value, a # character must be escaped to prevent it from starting a comment, but an = character does not require an escape. Consider these examples:

```
A=1           # Name is "A", value is "1"
B= 2 3 4      # Name is "B", value is "2 3 4"
C=5=\#6 # 7   # Name is "C", value is "5=#6"
```

Note that, two successive backslashes in a property value become a single backslash. To get two consecutive backslashes, you must escape each one with another backslash:

```
AServer=\\\\server\dir    # Value is "\\server\dir"
BServer=\\server\\dir     # Value is "\server\dir"
```

The preceding example also illustrates that, if a backslash is not followed by a backslash, #, or =, the backslash and the character following it are both preserved.

See Also

- Using Configuration Files
- Reading Properties
- Setting Properties on the Command Line

Setting Properties on the Command Line

In addition to setting properties in a configuration file, you can also set properties on the command line, for example:

```
server --Ice.UDP.SndSize=65535 --IceSSL.Trace.Security=2
```

Any command line option that begins with `--` and is followed by one of the reserved prefixes is read and converted to a property setting when you create a communicator. Property settings on the command line override settings in a configuration file. If you set the same property more than once on the same command line, the last setting overrides any previous ones.

For convenience, any property not explicitly set to a value is set to the value `1`. For example,

```
server --Ice.Trace.Protocol
```

is equivalent to

```
server --Ice.Trace.Protocol=1
```

Note that this feature only applies to properties that are set on the command line, but not to properties that are set from a configuration file.

You can also clear a property from the command line as follows:

```
server --Ice.Trace.Protocol=
```

As for properties set from a configuration file, assigning nothing to a property clears that property.

See Also

- Properties Overview
- Using Configuration Files

Using Configuration Files

The ability to configure an application's properties externally provides a great deal of flexibility: you can use any combination of command-line options and configuration files to achieve the desired settings, all without having to modify your application. This page describes two ways of loading property settings from a file.

On this page:

Prerequisites for Using Configuration Files

The Ice run time automatically loads a configuration file during the creation of a *property set*, which is an instance of the `Ice::Properties` interface. Every communicator has its own property set from which it derives its configuration. If an application does not supply a property set when it calls `Ice::initialize` (or the equivalent in other language mappings), the Ice run time internally creates a *property set* for the new communicator.

Note however that Ice loads a configuration file automatically only when the application creates a property set using an argument vector. This occurs when the application passes an argument vector to create a property set explicitly, or when the application passes an argument vector to `Ice::initialize`.

Both of the mechanisms described below can also retrieve property settings from additional sources.

The `ICE_CONFIG` Environment Variable

Ice automatically loads the contents of the configuration file named in the `ICE_CONFIG` environment variable (assuming the prerequisites ar e met). For example:

```
export ICE_CONFIG=/usr/local/filesystem/config
./server
```

This causes the server to read its property settings from the configuration file in `/usr/local/filesystem/config`.

If you use the `ICE_CONFIG` environment variable together with command-line options for other properties, the settings on the command line override the settings in the configuration file. For example:

```
export ICE_CONFIG=/usr/local/filesystem/config
./server --Ice.MessageSizeMax=4096
```

This sets the value of the `Ice.MessageSizeMax` property to `4096` regardless of any setting of this property in `/usr/local/filesystem /config`.

You can use multiple configuration files by specifying a list of configuration file names separated by commas. For example:

```
export ICE_CONFIG=/usr/local/filesystem/config,./config
./server
```

This causes property settings to be retrieved from `/usr/local/filesystem/config`, followed by any settings in the file `config` in the current directory; settings in `./config` override settings `/usr/local/filesystem/config`.

The `Ice.Config` Property

The `Ice.Config` property has special meaning to the Ice run time: it determines the path name of a configuration file from which to read property settings. For example:

```
./server --Ice.Config=/usr/local/filesystem/config
```

This causes property settings to be read from the configuration file in `/usr/local/filesystem/config`.

The `--Ice.Config` command-line option overrides any setting of the `ICE_CONFIG` environment variable, that is, if the `ICE_CONFIG` environment variable is set and you also use the `--Ice.Config` command-line option, the configuration file specified by the `ICE_CONFIG` environment variable is ignored.

If you use the `--Ice.Config` command-line option together with settings for other properties, the settings on the command line override the settings in the configuration file. For example:

```
./server --Ice.Config=/usr/local/filesystem/config --Ice.MessageSizeMax
=4096
```

This sets the value of the `Ice.MessageSizeMax` property to `4096` regardless of any setting of this property in `/usr/local/filesystem/config`. The placement of the `--Ice.Config` option on the command line has no influence on this precedence. For example, the following command is equivalent to the preceding one:

```
./server --Ice.MessageSizeMax=4096 --Ice.Config=/usr/local/filesystem/c
onfig
```

Settings of the `Ice.Config` property inside a configuration file are ignored, that is, you can set `Ice.Config` only on the command line.

If you use the `--Ice.Config` option more than once, only the last setting of the option is used and the preceding ones are ignored. For example:

```
./server --Ice.Config=file1 --Ice.Config=file2
```

This is equivalent to using:

```
./server --Ice.Config=file2
```

You can use multiple configuration files by specifying a list of configuration file names separated by commas. For example:

```
./server --Ice.Config=/usr/local/filesystem/config,./config
```

This causes property settings to be retrieved from `/usr/local/filesystem/config`, followed by any settings in the file `config` in the current directory; settings in `./config` override settings `/usr/local/filesystem/config`.

See Also

- Alternate Property Stores
- The Properties Interface

Alternate Property Stores

In addition to regular files, Ice also supports storing property settings in the Windows registry and Java resources.

On this page:

Loading Properties from the Windows Registry

You can use the Windows registry to store property settings. Property settings must be stored with a key underneath `HKEY_LOCAL_MACHINE`. To inform the Ice run time of this key, you must set the `Ice.Config` property to the key. For example:

```
client --Ice.Config=HKLM\MyCompany\MyApp
```

The Ice run time examines the value of `Ice.Config`; if that value begins with `HKLM`, the remainder of the property is taken to be a key to a number of string values. For the preceding example, the Ice run time looks for the key `HKEY_LOCAL_MACHINE\MyCompany\MyApp`. The string values stored under this key are used to initialize the properties.

The name of each string value is the name of the property (such as `Ice.Trace.Network`). Note that the value must be a string (even if the property setting is numeric). For example, to set `Ice.Trace.Network` to 3, you must store the string "3" as the value, not a binary or `DWORD` value.

String values in the registry can be regular strings (`REG_SZ`) or expandable strings (`REG_EXPAND_SZ`). Expandable strings allow you to include symbolic references to environment variables (such as `%ICE_HOME%`).

> Depending on whether you use 32-bit or 64-bit binaries, you must set the registry keys in the corresponding 32-bit or 64-bit registry. See http://support.microsoft.com/kb/305097 for more information.

Loading Properties from Java Resources

The Ice run time for Java supports the ability to load a configuration file as a class loader resource, which is especially useful for deploying an Ice application in a self-contained JAR file. For example, suppose we define `ICE_CONFIG` as shown below:

```
export ICE_CONFIG=app_config
```

During the creation of a property set (which often occurs implicitly when initializing a new communicator), Ice asks the Java run time to search the application's class path for a file named `app_config`. This file might reside in the same JAR file as the application's class files, or in a different JAR file in the class path, or it might be a regular file located in one of the directories in the class path. If Java is unable to locate the configuration file in the class path, Ice attempts to open the file in the local file system.

The class path resource always takes precedence over a regular file. In other words, if a class path resource and a regular file are both present with the same path name, Ice always loads the class path resource in preference to the regular file.

The path name for a class path resource uses a relative Unix-like format such as `subdir/myfile`. Java searches for the resource relative to each JAR file or subdirectory in an application's class path.

See Also

- Using Configuration Files

Command-Line Parsing and Initialization

On this page:

Parsing Command Line Options

When you initialize the Ice run time by calling `initialize`, you can pass the application's arguments to the initialization call.

In most language mappings, this argument vector is an *in-out* parameter. In C++, for example, `argc` is passed as a *reference* to an `int`:

C++11 C++98

```
std::shared_ptr<Ice::Communicator> initialize(int& argc, const char*
argv[], ...other parameters...);
```

```
Ice::CommunicatorPtr initialize(int& argc, const char* argv[], ...other
parameters...);
```

`initialize` parses the argument vector and initializes the new communicator's properties accordingly. It also removes all arguments that set Ice properties from the provided argument vector. For example, assume we invoke a C++ server as:

```
./server --myoption --Ice.Config=config -x a --Ice.Trace.Network=3 -y o
pt file
```

Initially, `argc` has the value 9, and `argv` has ten elements: the first nine elements contain the program name and the arguments, and the final element, `argv[argc]`, contains a null pointer (as required by the C++ standard). When `Ice::initialize` returns, `argc` has the value 7 and `argv` contains the following elements:

```
./server
--myoption
-x
a
-y
opt
file
0             # Terminating null pointer
```

This means that you should initialize the Ice run time before you parse the command line for your application-specific arguments. That way, the Ice-related options are stripped from the argument vector for you so you do not need to explicitly skip them.

`initialize` provides the same argument-property parsing and stripping in all language mappings.

If you use the Application helper class, the `run` member function or method is passed an argument vector with the Ice-related options already stripped. The same is true for the `runWithSession` member function or method called by the `Glacier2::Application` helper class.

The `Ice.ProgramName` Property

For C++, Objective-C, Python, and Ruby, `initialize` sets the `Ice.ProgramName` property to the name of the current program (`argv[0]`). In C#, `initialize` sets `Ice.ProgramName` to the value of `System.AppDomain.CurrentDomain.FriendlyName`. Your application code can read this property and use it for activities such as logging diagnostic or trace messages.

Even though `Ice.ProgramName` is initialized for you, you can still override its value from a configuration file or by setting the property on the command line.

For Java, the program name is not supplied as part of the argument vector — if you want to use the `Ice.ProgramName` property in your application, you must set it before initializing a communicator.

See Also

- Using Configuration Files
- Reading Properties
- Communicator Initialization
- Glacier2 Application Class

The Properties Interface

You can use the same configuration file and command-line mechanisms to set application-specific properties. For example, we could introduce a property to control the maximum file size for our file system application:

```
# Configuration file for file system application

Filesystem.MaxFileSize=1024    # Max file size in kB
```

The Ice run time stores the `Filesystem.MaxFileSize` property like any other property and makes it accessible via the `Properties` inter face.

To access property values from within your program, you need to acquire the communicator's properties by calling `getProperties`:

Slice

```
module Ice
{
    local interface Properties; // Forward declaration

    local interface Communicator
    {
        Properties getProperties();

        // ...
    }
}
```

The `Properties` interface is shown below:

Slice

```
module Ice
{
    local dictionary<string, string> PropertyDict;

    local interface Properties
    {
        string getProperty(string key);
        string getPropertyWithDefault(string key, string value);
        int getPropertyAsInt(string key);
        int getPropertyAsIntWithDefault(string key, int value);
        PropertyDict getPropertiesForPrefix(string prefix);

        void setProperty(string key, string value);

        StringSeq getCommandLineOptions();
        StringSeq parseCommandLineOptions(string prefix,
StringSeq options);
        StringSeq parseIceCommandLineOptions(StringSeq options);

        void load(string file);

        Properties clone();
    }
}
```

Most of the operations involve reading properties, setting properties, and parsing properties.

The `Properties` interface also provides two utility operations that are useful if you need to work with multiple communicators that use different property sets:

- `clone`
  This operation makes a copy of an existing property set. The copy contains exactly the same properties and values as the original.
- `load`
  This operation accepts a path name to a configuration file and initializes the property set from that file. If the specified file cannot be read (for example, because it does not exist or the caller does not have read permission), the operation throws a `FileException`. In Java, the given path name can refer to a class loader resource or a regular file.

See Also

- Using Configuration Files
- Reading Properties
- Setting Properties
- Parsing Properties
- Alternate Property Stores

Reading Properties

The `Properties` interface provides the following operations for reading property values:

- `string getProperty(string key)`
  This operation returns the value of the specified property. If the property is not set, the operation returns the empty string.
- `string getPropertyWithDefault(string key, string value)`
  This operation returns the value of the specified property. If the property is not set, the operation returns the supplied default value.
- `int getPropertyAsInt(string key)`
  This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns zero.
- `int getPropertyAsIntWithDefault(string key, int value)`
  This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns the supplied default value.
- `StringSeq getPropertyAsList(string key)`
  This operation returns the value as a list of strings. The strings must be separated by whitespace or a comma. If a string contains whitespace or a comma, it must be enclosed using single or double quotes; quotes can be themselves escaped in a quoted string, for example O'Reilly can be written as O'Reilly, "O'Reilly" or "O\'Reilly". If the property is not set or set to an empty value, the operation returns an empty list.
- `StringSeq getPropertyAsListWithDefault(string key, StringSeq value)`
  This operation returns the value as a list of strings (see `getPropertyAsList` above). If the property is not set or set to an empty value, the operation returns the supplied default value.
- `PropertyDict getPropertiesForPrefix(string prefix)`
  This operation returns all properties that begin with the specified prefix as a dictionary of type `PropertyDict`. This operation is useful if you want to extract the properties for a specific subsystem. For example,
  `getPropertiesForPrefix("Filesystem")`
  returns all properties that start with the prefix `Filesystem`, such as `Filesystem.MaxFileSize`. You can then use the usual dictionary lookup operations to extract the properties of interest from the returned dictionary.

With these operations, using application-specific properties now becomes the simple matter of initializing a communicator as usual, getting access to the communicator's properties, and examining the desired property value. For example:

C++11 Java

```
Ice::CommunicatorHolder ich(argc, argv);
// Get the maximum file size.
//
auto props = ich->getProperties(); // props is
std::shared_ptr<Ice::Properties>
int maxSize
= props->getPropertyAsIntWithDefault("Filesystem.MaxFileSize", 1024);
```

```
try(com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(args))
{
    // Get the maximum file size.
    //
    com.zeroc.Ice.Properties props = communicator.getProperties();
    int maxSize
= props.getPropertyAsIntWithDefault("Filesystem.MaxFileSize", 1024);
    ...
}
```

Assuming that you have created a configuration file that sets the `Filesystem.MaxFileSize` property (and that you have set the `ICE_CONFIG` variable or the `--Ice.Config` option accordingly), your application will pick up the configured value of the property.

> The technique shown above allows you to obtain application-specific properties from a *configuration file*. If you also want the ability to set application-specific properties on the command line, you will need to parse command-line options for your prefix. (Calling `initialize` to create a communicator only parses those command line options having a reserved prefix.)

See Also

- The Properties Interface
- Using Configuration Files
- Setting Properties
- Parsing Properties

Setting Properties

The `setProperty` operation on the `Properties` interface sets a property to the specified value:

Slice

```
module Ice
{
    local interface Properties
    {
        void setProperty(string key, string value);
        // ...
    }
}
```

You can clear a property by setting it to the empty string.

For properties that control the Ice run time and its services (that is, properties that start with one of the reserved prefixes), this operation is useful only if you call it *before* you call `initialize`. This is because property values are usually read by the Ice run time only once, when you call `initialize`, so the Ice run time does not pay attention to a property value that is changed after you have initialized a communicator. Of course, this begs the question of how you can set a property value and have it also recognized by a communicator.

To permit you to set properties before initializing a communicator, the Ice run time provides an overloaded helper function or method called `createProperties` that creates a property set.

`createProperties` is provided in all language mappings with overloads that accept the following parameters:

1. no parameter at all
   This `createProperties` creates an empty property set, and does *not* check `ICE_CONFIG` for a configuration file to parse.

2. the arguments given to the application plus a default property set
   This `createProperties` returns a property set that contains all the property settings that are passed as the default, plus any property settings in the arguments. If the argument vector sets a property that is also set in the passed default property set, the setting in the argument vector overrides the default. It also looks for the `--Ice.Config` option; if the argument vector specifies a configuration file, the configuration file is parsed. The order of precedence of property settings, from lowest to highest, is:
   - Property settings passed in the default parameter
   - Property settings set in the configuration file
   - Property settings in the argument vector (parsed as described in Command-Line Parsing and Initialization)

   This `createProperties` also looks for the setting of the `ICE_CONFIG` environment variable and, if that variable specifies a configuration file, parse that file. (However, an explicit `--Ice.Config` option in the argument vector or the `defaults` parameter overrides any setting of the `ICE_CONFIG` environment variable.)

3. the arguments given to the application
   This `initialize` is like the initialize in `2.` above, with an empty default property set.

C++11C++98C#JavaJava CompatJavaScriptMATLABObjCPHPPythonRuby

```
namespace Ice
{
    // (1): empty properties
    std::shared_ptr<Properties> createProperties();

    // (2) and (3): clone default properties parameter (if present) then
parse args
    std::shared_ptr<Properties> createProperties(int argc&, const char*
argv[], const std::shared_ptr<Properties>& = nullptr);
    std::shared_ptr<Properties> createProperties(StringSeq& args, const
std::shared_ptr<Properties>& defaults = nullptr);

#ifdef _WIN32
    // (2) and (3): clone default properties parameter (if present) then
parse args
    std::shared_ptr<Properties> createProperties(int argc&, const
wchar_t* argv[], const std::shared_ptr<Properties>& defaults = nullptr);
#endif
}
```

```
namespace Ice
{
    // (1): empty properties
    PropertiesPtr createProperties();

    // (2) and (3): clone default properties parameter (if present) then
parse args
    PropertiesPtr createProperties(int argc&, const char* argv[], const
PropertiesPtr& = 0);
    PropertiesPtr createProperties(StringSeq& args, const PropertiesPtr&
defaults = 0);

#ifdef _WIN32
    // (2) and (3): clone default properties parameter (if present) then
parse args
    PropertiesPtr createProperties(int argc&, const wchar_t* argv[],
const PropertiesPtr& defaults = 0);
#endif
}
```

```
namespace Ice
{
    public sealed class Util
    {
        // (1): empty properties
        public static Properties createProperties() { ... }

        // (2): clone default properties and then parse args
        public static Properties
createProperties(ref string[] args, Properties defaults) { ... }

        // (3): parse args
        public static Properties createProperties(ref string[] args) {
... }
    }
}
```

```
package com.zeroc.Ice;

public final class Util
{
    // (1): empty properties
    public static Properties createProperties() { ... }

    // (2): clone default properties and then parse args
    public static Properties createProperties(String[] args, Properties
defaults)  { ... }
    public static Properties createProperties(String[] args, Properties
defaults, java.util.List<String> remainingArgs)  { ... }
    // (3): parse args
    public static Properties createProperties(String[] args) { ... }
    public static Properties createProperties(String[] args,
java.util.List<String> remainingArgs)  { ... }

    // ...
}
```

```
package Ice;

public final class Util
{
    // (1): empty properties
    public static Properties createProperties() { ... }
    // (2): clone default properties and then parse args
    public static Properties createProperties(String[] args, Properties
defaults) { ... }
    public static Properties createProperties(StringSeqHolder args,
Properties defaults) { ... }

    // (3): parse args
    public static Properties createProperties(String[] args) { ... }
    public static Properties createProperties(StringSeqHolder args) {
... }

    // ...
}
```

```
Ice.createProperties = function(args, defaults) { ... }
```

```
% in Ice package

% Valid arguments:
%
% (1): no parameter at all, returns empty properties
% props = Ice.createProperties();
% (2): clone default properties and then parse args (a cell array of
character vectors)
% [props, remainingArgs] = createProperties(args, defaults);
% (3): parse args (a cell array of character vectors)
% [props, remainingArgs] = createProperties(args);

function [props, remainingArgs] = createProperties(varargin)
```

```
@interface ICEUtil : NSObject

// (1): empty properties
+(id<ICEProperties>) createProperties;

// (3): parse args
+(id<ICEProperties>) createProperties:(int*)argc argv:(char*[])argv;
...
@end
```

```
namespace Ice
{
    function createProperties(args=array(), defaults=null) { ... }
}
```

```
# in Ice module
def createProperties(args=[], defaults=None)
```

```
module Ice
    def createProperties(args=[], defaults=None)
```

Like `initialize`, `createProperties` strips Ice-related command-line options from the passed argument vector.

`createProperties` is useful if you want to ensure that a property is set to a particular value, regardless of any setting of that property in a configuration file or in the argument vector. Here is an example:

C++11 Java MATLAB PHP Python Ruby

```
// Get the initialized property set.
//
auto props = Ice::createProperties(argc, argv);

// Make sure that network and protocol tracing are off.
//
props->setProperty("Ice.Trace.Network", "0");
props->setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
Ice::InitializationData initData;
initData.properties = props;
Ice::CommunicatorHolder  ich(initData);
```

```
// Get the initialized property set.
//
com.zeroc.Ice.Properties props =
com.zeroc.Ice.Util.createProperties(args);

// Make sure that network and protocol tracing are off.
//
props.setProperty("Ice.Warn.Connections", "0");
props.setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
com.zeroc.Ice.InitializationData initData = new
com.zeroc.Ice.InitializationData();
initData.properties = props;
com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(initData);
```

```
% Get the initialized property set.
%
props = Ice.createProperties(args);

% Make sure that network and protocol tracing are off.
%
props.setProperty('Ice.Warn.Connections', '0');
props.setProperty('Ice.Trace.Protocol', '0');

% Initialize a communicator with these properties.
%
initData = Ice.InitializationData();
initData.properties_ = props;
communicator = Ice.initialize(initData);
```

```
// Get the initialized property set.
//
$props = Ice\createProperties($args);

// Make sure that network and protocol tracing are off.
//
$props->setProperty("Ice.Trace.Network", "0");
$props->setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
$initData = new Ice\InitializationData();
$initData->properties = $props;
$communicator = Ice\initialize($initData);
```

```
# Get the initialized property set.
#
props = Ice.createProperties(sys.argv)

# Make sure that network and protocol tracing are off.
#
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")

# Initialize a communicator with these properties.
#
initData = Ice.InitializationData()
initData.properties = props
communicator = Ice.initialize(initData)
```

```
# Get the initialized property set.
#
props = Ice::createProperties(ARGV)

# Make sure that network and protocol tracing are off.
#
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")

# Initialize a communicator with these properties.
#
initData = Ice::InitializationData.new
initData.properties = props
initData = Ice::initialize(initData)
```

You can also set properties "in bulk" remotely through the Properties facet of Ice Admin.

See Also

- The Properties Interface
- Using Configuration Files
- Command-Line Parsing and Initialization
- Reading Properties
- Parsing Properties
- Communicator Initialization

Parsing Properties

The `Properties` interface provides several operations for converting properties to and from command-line options.

On this page:

Converting Properties to Command-Line Options

The `getCommandLineOptions` operation converts an initialized set of properties into a sequence of equivalent command-line options:

Slice

```
module Ice
{
    local interface Properties
    {
        StringSeq getCommandLineOptions();
        // ...
    }
}
```

For example, if you have set the `Filesystem.MaxFileSize` property to 1024 and call `getCommandLineOptions`, the setting is returned as the string `"--Filesystem.MaxFileSize=1024"`. This operation is useful for diagnostic purposes, for example, to dump the setting of all properties to a logging facility, or if you want to fork a new process with the same property settings as the current process.

Converting Command-Line Options to Properties

The `parseCommandLineOptions` operation examines the passed argument vector for command-line options that have the specified prefix:

Slice

```
module Ice
{
    local interface Properties
    {
        StringSeq parseCommandLineOptions(string prefix, StringSeq
options);
        // ...
    }
}
```

All options having the form `--`*`prefix`*`.Key=Value` are converted to property settings (that is, they initialize the corresponding properties). The operation returns an argument vector containing the options that did not match the prefix.

The value for *`prefix`* has an implicit trailing period if one is not present. For example, when calling `parseCommandLineOptions` with a prefix value of `"File"`, the option `--File.Owner=root` would match but the option `--Filesystem.MaxFileSize=1024` would not match.

The operation parses command-line options using the same syntax rules as for properties in a configuration file. However, the user's command shell can cause differences in parsing behavior. Suppose we define the following property in a configuration file:

```
MyApp.Home=C:\Program Files\MyApp
```

The presence of whitespace in the property definition is not an issue in a configuration file but can be an issue on the command line, where the equivalent option is `--MyApp.Home=C:\Program Files\MyApp`. If the user is not careful, the program may receive this as two separate options: `--MyApp.Home=C:\Program` and `Files\MyApp`. In the end, it is the user's responsibility to ensure that the property's complete key and value are contained within a single command-line option.

C++ Helper Functions

Because `parseCommandLineOptions` expects a sequence of strings while C++ programs are used to dealing with `argc` and `argv`, Ice provides two utility functions that convert an `argc`/`argv` vector into a sequence of strings and vice-versa:

C++

```
namespace Ice
{
    // Ice::StringSeq is a std::vector<std::string>
    //
    StringSeq argsToStringSeq(int argc, const char* const argv[]);

    void stringSeqToArgs(const StringSeq& args, int& argc, const
char* argv[]);
    inline void stringSeqToArgs(const StringSeq& seq, int& argc, char*
argv[])
    {
        // adapts the argv parameter
        return stringSeqToArgs(seq, argc, const_cast<const
char**>(argv));
    }
}
```

You need to use `parseCommandLineOptions` (and the utility functions) if you want to permit application-specific properties to be set from the command line. For example, to allow the `--Filesystem.MaxFileSize` option to be used on the command line, we need to initialize our program as follows:

C++11 C++98

```
int
main(int argc, char* argv[])
{
    // Create an empty property set.
    //
    auto props = Ice::createProperties();

    // Convert argc/argv to a string sequence.
    //
    auto args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Convert remaining arguments back to argc/argv vector.
    //
    Ice::stringSeqToArgs(args, argc, argv);

    // Initialize communicator.
    //
    Ice::InitializationData initData;
    initData.properties = props;
    Ice::CommunicatorHolder ich(argc, argv, id);

    // At this point, argc/argv only contain options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

```
int
main(int argc, char* argv[])
{
    // Create an empty property set.
    //
    Ice::PropertiesPtr props = Ice::createProperties();

    // Convert argc/argv to a string sequence.
    //
    Ice::StringSeq args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Convert remaining arguments back to argc/argv vector.
    //
    Ice::stringSeqToArgs(args, argc, argv);

    // Initialize communicator.
    //
    Ice::InitializationData initData;
    initData.properties = props;
    Ice::CommunicatorHolder ich(argc, argv, id);

    // At this point, argc/argv only contain options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

Using this code, any options beginning with `--Filesystem` are converted to properties and are available via the property lookup operations as usual. The call to `initialize` made by the `CommunicatorHolder` constructor removes any Ice-specific command-line options so, once the communicator is created, `argc`/`argv` only contains options and arguments that are not related to setting either a filesystem or an Ice property.

An easier way to achieve the same thing is to use a `CommunicatorHolder` constructor that accepts a string sequence, instead of an `argc`/`argv` pair:

C++11 C++98

```
int
main(int argc, char* argv[])
{
    // Create an empty property set.
    //
    auto props = Ice::createProperties();

    // Convert argc/argv to a string sequence.
    //
    auto args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Initialize communicator.
    //
    Ice::InitializationData initData;
    initData.properties = props;
    Ice::CommunicatorHolder ich(args, initData);

    // At this point, args only contains options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

```
int
main(int argc, char* argv[])
{
    // Create an empty property set.
    //
    Ice::PropertiesPtr props = Ice::createProperties();

    // Convert argc/argv to a string sequence.
    //
    Ice::StringSeq args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Initialize communicator.
    //
    Ice::InitializationData initData;
    initData.properties = props;
    Ice::CommunicatorHolder ich(args, initData);

    // At this point, args only contains options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

Converting Reserved Command-Line Options to Properties

The `parseIceCommandLineOptions` operation behaves like `parseCommandLineOptions`, but removes the reserved Ice-specific options from the argument vector:

Slice

```
module Ice
{
    local interface Properties
    {
        StringSeq parseIceCommandLineOptions(StringSeq options);
        // ...
    }
}
```

This operation is also used internally by the Ice run time to parse Ice-specific options in `initialize`.

See Also

- Properties Overview
- The Properties Interface
- Reading Properties
- Setting Properties
- Command-Line Parsing and Initialization
- Logger Facility

Communicator and other Core Local Features

This section presents several core features of the Ice run-time, that are purely local to your client or server application.

Topics

Communicator

The main entry point to the Ice run time is represented by the local Slice interface `Ice::Communicator`. An instance of `Ice::Communicator` is associated with a number of run-time resources:

- Client-side thread pool
  The client-side thread pool is used to process replies to asynchronous method invocations (AMI), to avoid deadlocks in callbacks, and to process incoming requests on bidirectional connections.
- Server-side thread pool
  Threads in this pool accept incoming connections and handle requests from clients.
- Configuration properties
  Various aspects of the Ice run time can be configured via properties. Each communicator has its own set of such configuration properties.
- Object factories
  In order to instantiate classes that are derived from a known base type, the communicator maintains a set of object factories that can instantiate the class on behalf of the Ice run time. Object factories are discussed in each client-side language mapping.
- Logger object
  A logger object implements the `Ice::Logger` interface and determines how log messages that are produced by the Ice run time are handled.
- Default router
  A router implements the `Ice::Router` interface. Routers are used by Glacier2 to implement the firewall functionality of Ice.
- Default locator
  A locator is an object that resolves an object identity to a proxy. A locator object is implemented by a location service.
- Plug-in manager
  Plug-ins are objects that add features to a communicator. For example, IceSSL is implemented as a plug-in. Each communicator has a plug-in manager that implements the `Ice::PluginManager` interface and provides access to the set of plug-ins for a communicator.
- Object adapters
  Object adapters dispatch incoming requests and take care of passing each request to the correct servant.

Object adapters and objects that use different communicators are completely independent from each other. Specifically:

- Each communicator uses its own thread pool. This means that if, for example, one communicator runs out of threads for incoming requests, only objects using that communicator are affected. Objects using other communicators have their own thread pool and are therefore unaffected.
- Collocated invocations across different communicators are not optimized, whereas collocated invocations using the same communicator bypass much of the overhead of call dispatch.

Typically, servers use only a single communicator but, occasionally, multiple communicators can be useful. For example, IceBox, uses a separate communicator for each Ice service it loads to ensure that different services cannot interfere with each other. Multiple communicators are also useful to avoid thread starvation: if one service runs out of threads, this leaves the remaining services unaffected.

See Also

- Communicator Shutdown and Destruction
- Obtaining Proxies
- Converting Proxies to Strings
- Creating an Object Adapter
- Object Identity
- Value Factories
- Properties and Configuration
- Implicit Request Contexts
- Plug-in Facility
- Logger Facility
- Locators
- Routers
- Logger Facility
- Administrative Facility
- Instrumentation Facility

Communicator Initialization

On this page:

Creating a Communicator with `initialize`

The first step for any Ice application is to initialize the Ice run time by calling the Ice `initialize` native function or method. (`initialize` is named `createCommunicator` in Objective-C). `initialize` returns a `Communicator` object, which represents an instance of the Ice run time.

You can call `initialize` directly (all language mappings) or in C++ through the `Ice::CommunicatorHolder` helper class.

Each language mapping provides several overloads for `initialize` with the following parameters:

1. the arguments given to the application
   This `initialize` extracts Ice properties from the provided arguments, and sets them in the newly created communicator. See Command-Line Parsing and Initialization for further details.

2. the arguments given to the application plus an `InitializationData` parameter
   This `initialize` uses the `InitializationData` parameter (described below on this page) to create a new communicator, and then extracts Ice properties from the provided arguments. The properties extracted from the arguments overwrite properties set in `InitializationData`.

3. the arguments given to the application plus a config file parameter
   This `initialize` is a shortcut for the `initialize` in 2. above, where the `InitializationData` parameter contains only properties loaded from the provided configuration file. For example, here is the corresponding implementation in Java:

   Java

   ```
   public static Communicator initialize(String[] args, String
   configFile, java.util.List<String> remainingArgs)
   {
       InitializationData initData = null;
       if(configFile != null)
       {
           initData = new InitializationData();
           initData.properties = Util.createProperties();
           initData.properties.load(configFile);
       }
       return initialize(args, initData, remainingArgs);
   }
   ```

4. an `InitializationData` parameter
   This `initialize` uses the `InitializationData` parameter (described below on this page) to create a new communicator. It's similar to `initialize` in 2. above, except this `initialize` does not take arguments and does not check the `ICE_CONFIG` environment variable.

5. a config file parameter
   This `initialize` is a shortcut for the `initialize` in 4. above, where the `InitializationData` parameter contains only properties loaded from the provided configuration file.

6. no parameters at all
   This `initialize` creates a communicator with no properties (meaning all Ice properties have their default value), and only default features. It also does not check the `ICE_CONFIG` environment variable.

C++11 C++98 C# Java Java Compat JavaScript MATLAB ObjC PHP Python Ruby

```
namespace Ice
{
    // (1) and (2): argc/argv + InitializationData
    std::shared<Communicator> initialize(int& argv, const char* argv[],
const InitializationData& initData = InitializationData(),
                                         int = ICE_INT_VERSION);

    // (3): argc/argv + config file
    std::shared<Communicator> initialize(int& argc, const char* argv[],
const std::string& configFile,
                                         int version = ICE_INT_VERSION);

#ifdef _WIN32
    // (1) and (2): wide argc/argv + InitializationData
    std::shared<Communicator> initialize(int& argc, const wchar_t*
argv[], const InitializationData& initData = InitializationData(),
                                         int version = ICE_INT_VERSION);

    // (3): wide argc/argv + config file
    std::shared<Communicator> initialize(int& argc, const wchar_t*
argv[], const std::string& configFile,
                                         int version = ICE_INT_VERSION);
#endif

    // (1) and (2): args as a vector<string> + InitializationData
    std::shared<Communicator> initialize(StringSeq& args, const
InitializationData& initData = InitializationData(),
                                         int version = ICE_INT_VERSION);

    // (3): args as a vector<string> + config file
    std::shared<Communicator> initialize(StringSeq& args, const
std::string& configFile,
                                         int version = ICE_INT_VERSION);

    // (4) and (6): no args + InitializationData
    std::shared<Communicator> initialize(const InitializationData&
initData = InitializationData(),
                                         int version = ICE_INT_VERSION);

    // (5) no args + config file
    std::shared<Communicator> initialize(const std::string& configFile,
                                         int version= ICE_INT_VERSION);
}
```

All `initialize` functions have a trailing parameter `int version = ICE_INT_VERSION`. This parameter ensures the Ice version in the header files you used to compile your application is compatible with the Ice version in the Ice libraries loaded by this

application. You should always keep the default value (`ICE_INT_VERSION`) for this parameter.

```
namespace Ice
{
    // (1) and (2): argc/argv + InitializationData
    CommunicatorPtr initialize(int& argv, const char* argv[], const
InitializationData& initData = InitializationData(),
                               int = ICE_INT_VERSION);

    // (3): argc/argv + config file
    CommunicatorPtr initialize(int& argc, const char* argv[], const
char* configFile,
                               int version = ICE_INT_VERSION);

#ifdef _WIN32
    // (1) and (2): wide argc/argv + InitializationData
    CommunicatorPtr initialize(int& argc, const wchar_t* argv[], const
InitializationData& initData = InitializationData(),
                               int version = ICE_INT_VERSION);

    // (3): wide argc/argv + config file
    CommunicatorPtr initialize(int& argc, const wchar_t* argv[], const
char* configFile,
                               int version = ICE_INT_VERSION);
#endif

    // (1) and (2): args as a vector<string> + InitializationData
    CommunicatorPtr initialize(StringSeq& args, const
InitializationData& initData = InitializationData(),
                               int version = ICE_INT_VERSION);

    // (3): args as a vector<string> + config file
    CommunicatorPtr initialize(StringSeq& args, const char* configFile,
                               int version = ICE_INT_VERSION);

    // (4) and (6): no args + optional InitializationData
    CommunicatorPtr initialize(const InitializationData& initData =
InitializationData(),
                               int version = ICE_INT_VERSION);

    // (5): no args + config file
    CommunicatorPtr initialize(const char* configFile,
                               int version= ICE_INT_VERSION);
}
```

All `initialize` functions have a trailing parameter `int version = ICE_INT_VERSION`. This parameter ensures the Ice version in the header files you used to compile your application is compatible with the Ice version in the Ice libraries loaded by this application. You should always keep the default value (`ICE_INT_VERSION`) for this parameter.

```
namespace Ice
{
    public sealed class Util
    {
        // (1): args only
        public static Communicator initialize(ref string[] args) { ... }

        // (2): args + InitializationData
        public static Communicator initialize(ref string[] args,
InitializationData initData) { ... }

        // (3): args + config file
        public static Communicator initialize(ref string[] args, string
configFile) { ... }

        // (4): no args + InitializationData
        public static Communicator initialize(InitializationData
initData) { ... }

        // (5): no args + config file
        public static Communicator initialize(string configFile) { ... }

        // (6): no parameter at all
        public static Communicator initialize() { ... }

        ...
     }
}
```

```
package com.zeroc.Ice;

public final class Util
{
    // (1): args only
    public static Communicator initialize(String[] args) { ... }
    public static Communicator initialize(String[] args,
java.util.List<String> remainingArgs) { ... }

    // (2): args + InitializationData
    public static Communicator initialize(String[] args,
InitializationData initData) { ... }
    public static Communicator initialize(String[] args,
InitializationData initData, java.util.List<String> remainingArgs) { ...
}

    // (3): args + config file
    public static Communicator initialize(String[] args, String
configFile) { ... }
    public static Communicator initialize(String[] args, String
configFile, java.util.List<String> remainingArgs) { ... }

    // (4): no args + InitializationData
    public static Communicator initialize(InitializationData initData) {
... }

    // (5): no args + config file
    public static Communicator initialize(String configFile) { ... }

    // (6): no parameter at all
    public static Communicator initialize() { ... }
    ...
}
```

```
package Ice;

public final class Util
{
    // (1): args only
    public static Communicator initialize(String[] args) { ... }
    public static Communicator initialize(StringSeqHolder args) { ... }

    // (2): args + InitializationData
    public static Communicator initialize(String[] args,
InitializationData initData) { ... }
    public static Communicator initialize(StringSeqHolder args,
InitializationData initData) { ... }

    // (3): args + config file
    public static Communicator initialize(String[] args, String
configFile) { ... }
    public static Communicator initialize(StringSeqHolder args, String
configFile) { ... }

    // (4): no args + InitializationData
    public static Communicator initialize(InitializationData initData) {
... }

    // (5) no args + config file
    public static Communicator initialize(String configFile) { ... }

    // (6): no parameter at all
    public static Communicator initialize() { ... }
    ...
}
```

```
// (1): args
// (2): args + InitializationData
// (4): InitializationData
// (6): no parameter at all
Ice.initialize = function(arg1, arg2)
```

```
% in Ice package

% initialize returns the new communicator and the remaining args

% (1): args (a cell array of character vectors)
% (2): args + InitializationData
% (3): args + config file
% (4): InitializationData
% (5): config file
% (6): no parameter at all
function [communicator, remainingArgs] = initialize(varargin)
```

```
@interface ICEUtil : NSObject
// (1): argc/argv only
+(id<ICECommunicator>) createCommunicator:(int*)argc argv:(char*[])argv;

// (2): argc/argv + ICEInitializationData
+(id<ICECommunicator>) createCommunicator:(int*)argc argv:(char*[])argv
initData:(ICEInitializationData*)initData;

// (3): argc/argv + config file
+(id<ICECommunicator>) createCommunicator:(int*)argc argv:(char*[])argv
configFile:(NSString*)configFile;

// (4): no args + InitializationData
+(id<ICECommunicator>)
createCommunicator:(ICEInitializationData*)initData;

// (5) not provided in Objective-C due to overload ambiguity
// (6): no parameter at all
+(id<ICECommunicator>) createCommunicator;
...
@end
```

```
namespace Ice
{
    // (1): args
    // (2): args + InitializationData
    // (4): InitializationData
    // (6): no parameter at all
    function initialize(args=null, initData=null)
}
```

```
# in Ice module

# (1): args
# (2): args + InitializationData
# (3): args + config file
# (4): InitializationData
# (5): config file
# (6): no parameter at all
def initialize(args=None, data=None)
```

```
module Ice
    # (1): args
    # (2): args + InitializationData
    # (3): args + config file
    # (4): InitializationData
    # (5): config file
    # (6): no parameter at all
    def Ice.initialize(args=nil, initData=nil)
```

The `initialize` function optionally accepts a block.

`InitializationData`

During the creation of a communicator, the Ice run time initializes a number of features that affect the communicator's operation. Once set, these features remain in effect for the life time of the communicator, that is, you cannot change these features after you have created a communicator. Therefore, if you want to customize these features, you must do so when you create the communicator.

The following features can be customized at communicator creation time:

- the property set
- the logger object
- the instrumentation observer
- the thread start and stop notification hooks
- the dispatcher
- the compact ID resolver (used by the streaming interfaces when extracting Ice objects)
- the class loader (Java only)
- the batch request interceptor
- the value factory manager

To establish these features, you initialize a structure or class of type `InitializationData`, set one or more fields of this structure or class, and pass this `InitializationData` to `initialize` or to a helper class (such as Ice `Application`) that calls `initialize` for you.

The exact definition of `InitializationData` depends on the language mapping you use, and some language mappings only support a subset of these features:

C++11C++98C#JavaJava CompatJavaScriptMATLABObjCPHPPythonRuby

```
namespace Ice
{
    struct InitializationData
    {
        std::shared_ptr<Ice::Properties> properties;
        std::shared_ptr<Ice::Logger> logger;
        std::shared_ptr<Ice::Instrumentation::CommunicatorObserver>
observer;
        std::function<void()> threadStart;
        std::function<void()> threadStop;
        std::function<void(std::function<void()>, const
std::shared_ptr<Ice::Connection>&)> dispatcher;
        std::function<std::string(int)> compactIdResolver;
        std::function<void(const Ice::BatchRequest&, int, int)>
batchRequestInterceptor;
        std::shared_ptr<Ice::ValueFactoryManager> valueFactoryManager;
    };
}
```

```
namespace Ice
{
    struct InitializationData
    {
        PropertiesPtr properties;
        LoggerPtr logger;
        Instrumentation::CommunicatorObserverPtr observer;
        ThreadNotificationPtr threadHook;
        DispatcherPtr dispatcher;
        CompactIdResolverPtr compactIdResolver;
        BatchRequestInterceptorPtr batchRequestInterceptor;
        ValueFactoryManagerPtr valueFactoryManager;
    };
}
```

```
namespace Ice
{
    public class InitializationData : ICloneable
    {
        public object Clone() { ... }

        public Properties properties;
        public Logger logger;
        public Instrumentation.CommunicatorObserver observer;
        public System.Action threadStart;
        public System.Action threadStop;
        public System.Action<System.Action, Connection> dispatcher;
        public System.Func<int, string> compactIdResolver;
        public System.Action<BatchRequest, int, int>
batchRequestInterceptor;
        public ValueFactoryManager valueFactoryManager;
    }
}
```

```
package com.zeroc.Ice;

public final class InitializationData implements Cloneable
{
    @Override
    public InitializationData clone() { ... }

    public Properties properties;
    public Logger logger;
    public com.zeroc.Ice.Instrumentation.CommunicatorObserver observer;
    public Runnable threadStart;
    public Runnable threadStop;
    public ClassLoader classLoader;
    public java.util.function.BiConsumer<Runnable, Connection>
dispatcher;
    public java.util.function.IntFunction<String> compactIdResolver;
    public BatchRequestInterceptor batchRequestInterceptor;
    public ValueFactoryManager valueFactoryManager;
}
```

```
package Ice;

public final class InitializationData implements Cloneable
{
    @Override
    public InitializationData clone() { ... }

    public Properties properties;
    public Logger logger;
    public Ice.Instrumentation.CommunicatorObserver observer;
    public ThreadNotification threadHook;
    public ClassLoader classLoader;
    public Dispatcher dispatcher;
    public CompactIdResolver compactIdResolver;
    public BatchRequestInterceptor batchRequestInterceptor;
    public ValueFactoryManager valueFactoryManager;
}
```

```
Ice.InitializationData = function()
{
    this.properties = null;
    this.logger = null;
    this.valueFactoryManager = null;
};
```

```
% in Ice package

classdef (Sealed) InitializationData
    properties
        properties_
    end
end
```

```
@interface ICEInitializationData : NSObject
{
...
}
@property(retain, nonatomic) id<ICEProperties> properties;
@property(retain, nonatomic) id<ICELogger> logger;
@property(copy, nonatomic) void(^dispatcher)(id<ICEDispatcherCall>,
id<ICEConnection>);
@property(copy, nonatomic)
void(^batchRequestInterceptor)(id<ICEBatchRequest>, int, int);

-(id) init:(id<ICEProperties>)properties logger:(id<ICELogger>)logger

dispatcher:(void(^)(id<ICEDispatcherCall>, id<ICEConnection>))d

batchRequestInterceptor:(void(^)(id<ICEBatchRequest>, int, int))i;
+(id) initializationData;
+(id) initializationData:(id<ICEProperties>)properties
logger:(id<ICELogger>)logger

dispatcher:(void(^)(id<ICEDispatcherCall>, id<ICEConnection>))d

batchRequestInterceptor:(void(^)(id<ICEBatchRequest>, int, int))i;
// This class also overrides copyWithZone:, hash, isEqual:, and dealloc.
@end
```

```
namespace Ice
{
    class InitializationData
    {
        public function __construct($properties=null, $logger=null) {
... }

        public $properties;
        public $logger;
    }
}
```

```
# in Ice module

class InitializationData(object):
    def __init__(self):
        self.properties = None
        self.logger = None
        self.threadStart = None
        self.threadStop = None
        self.dispatcher = None
        self.batchRequestInterceptor = None
```

```
module Ice
    class InitializationData
        def initialize(properties=nil, logger=nil)
            ...
        end

        attr_accessor :properties, :logger
    end
end
```

For example, to set a custom logger of type `MyLogger` in C++, you could use:

C++11 C++98

```
Ice::InitializationData initData;
initData.logger = std::make_shared<MyLoggerI>();
auto communicator = Ice::initialize(argc, argv, initData);
```

or

```
Ice::InitializationData initData;
initData.logger = std::make_shared<MyLoggerI>();
Ice::CommunicatorHolder ich(argc, argv, initData);
```

```
Ice::InitializationData initData;
initData.logger = new MyLoggerI;
Ice::CommunicatorPtr
communicator = Ice::initialize(argc, argv, initData);
```

or

```
Ice::InitializationData initData;
initData.logger = new MyLoggerI;
Ice::CommunicatorHolder ich(argc, argv, initData);
```

See Also

- Command-Line Parsing and Initialization
- The Properties Interface

Communicator Shutdown and Destruction

On this page:

Communicator Operations

Once you have successfully created a communicator, it is essential that you `destroy` this communicator before exiting your application. Destroying a communicator ensures an orderly closure of connections opened by this communicator, the clean termination of all the threads started by this communicator and more.

A communicator provides four operations related to shutdown and destruction:

Slice

```
["clr:implements:_System.IDisposable",
"java:implements:java.lang.AutoCloseable"]
local interface Communicator
{
    ["cpp:noexcept"] void shutdown();
    ["cpp:noexcept"] void waitForShutdown();
    ["cpp:noexcept"] bool isShutdown();
    ["cpp:noexcept"] void destroy();
    // ...
}
```

- `shutdown`
  This operation shuts down the server side of the Ice run time:
  - Operation invocations that are in progress at the time `shutdown` is called are allowed to complete normally. `shutdown` doe s *not* wait for these operations to complete; when `shutdown` returns, you know that no new incoming requests will be dispatched, but operations that were already in progress at the time you called `shutdown` may still be running. You can wait for still-executing operations to complete by calling `waitForShutdown`.
  - Operation invocations that arrive after the server has called `shutdown` either fail with a `ConnectFailedException` or are transparently redirected to a new instance of the server (via IceGrid).
  - Note that `shutdown` initiates deactivation of all object adapters associated with the communicator, so attempts to use an adapter once `shutdown` has completed raise an `ObjectAdapterDeactivatedException`.

  `shutdown` does not throw any exception, and calling `shutdown` multiple times on the same communicator is perfectly safe and correct.

- `waitForShutdown`
  On the server side, this operation suspends the calling thread until the communicator has shut down (that is, until no more operations are executing in the server). This allows you to wait until the server is idle before you destroy the communicator.
  On the client side, `waitForShutdown` simply waits until another thread has called `shutdown` or `destroy`.

- `isShutdown`
  This operation returns true if `shutdown` has been invoked on the communicator. A return value of true does not necessarily indicate that the shutdown process has completed, only that it has been initiated. An application that needs to know whether shutdown is complete can call `waitForShutdown`. If the blocking nature of `waitForShutdown` is undesirable, the application can invoke it from a separate thread.

- `destroy`
  This operation destroys the communicator and all its associated resources, such as threads, communication endpoints, object adapters, and memory resources. Once you have destroyed the communicator (and therefore destroyed the run time for that communicator), you must not call any other Ice operation (other than to create another communicator). `destroy` does not throw any exception, and calling `destroy` multiple times on the same communicator is perfectly safe and correct.
  If you call `destroy` without calling `shutdown`, the call waits for all executing operation invocations to complete before it returns

(that is, the implementation of `destroy` implicitly calls `shutdown` followed by `waitForShutdown`). `shutdown` (and, therefore, `destroy`) deactivates all object adapters that are associated with the communicator. Since `destroy` blocks until all operation invocations complete, a servant will deadlock if it invokes `destroy` on its own communicator while executing a dispatched operation.

On the client side, calling `destroy` while operations are still executing causes those operations to terminate with a `CommunicatorDestroyedException`.

Most language mappings provide constructs to `destroy` the communicator automatically when leaving a scope:

- C++: the `Ice::CommunicatorHolder` RAII helper class
- C#: `Communicator` implements the `IDisposable` interface, which allows you to initialize a communicator in a `using` statement
- Java: `Communicator` implements `java.lang.AutoCloseable`, which allows you to initialize a communicator in a try-with-resources statement
- Python: `Communicator` implements the Python context manager protocol, which allows you to call `initialize` in a `with` statement
- Ruby: `Ice::initialize` accepts an optional block, which destroys the communicator automatically at the end of the block

Common Patterns

Pure Client

In a pure client - a client that does not create any object adapter - there is no need to shutdown your communicator so you typically `destroy` your communicator without an intermediate explicit `shutdown`.

Server

In a server, you can destroy your communicator or first shut it down and only later destroy it.

- **Plain Destroy**
  In a server, a communicator `destroy` with no prior shutdown is correct provided this `destroy` is *not* called from an operation dispatch, and you don't need the communicator afterwards. For example, many C# and Java servers in ice-demos destroy their communicators in response to a CTRL-C or similar signal. This is the simplest signal handling strategy since C# (by default) and Java (always) terminate the application once the signal handlers have finished execution.

- **Shutdown with WaitForShutdown and Destroy**
  You can initiate the shutdown of your communicator in one thread (with `shutdown`) while another thread is blocked on `waitForShutdown`. When the thread blocked on `waitForShutdown` is freed, it has a shutdown but not destroyed communicator, still usable for remote invocations. This thread is then typically responsible to `destroy` the communicator.
  For example, all the ice-demos applications use this pattern to implement remote shutdown: the shutdown operation implementation calls `shutdown` on the communicator, while the main thread waits on `waitForShutdown` and later destroys the communicator.
  C++, Python and some C# demo servers also shutdown their communicator in response to a CTRL-C signal or similar signal. With C++, Python and optionally in C#, the application keeps running after a signal is handled, so we can have the main thread perform post-shutdown cleanups (if needed) and then destroy the communicator before completing cleanly.

Application Helper Class

On this page:

The Ice `Application` Helper Class

Ice provides an `Application` helper class for most programming languages. This helper class initializes the Ice run time then runs your application's code. It also makes sure the Ice run time is properly finalized when an exception is thrown or when the application receives a signal such a keyboard interrupt.

> `Application` used to be a staple of all Ice demo applications and was shown in many examples in this Manual. We have since updated the demos and examples in this Manual to use native language constructs instead. While `Application` certainly still works, we no longer recommend using `Application` for new applications.

Application is defined as follows (with some detail omitted for now):

C++11C++98C#JavaJava CompatPythonRuby

```
namespace Ice
{
    enum class SignalPolicy : unsigned char { HandleSignals,
NoSignalHandling };

    class Application
    {
    public:

        Application(SignalPolicy = SignalPolicy::HandleSignals);
        virtual ~Application();

        int main(int argc, const char* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
        int main(int argc, const char* const argv[], const std::string&
configFile, int version = ICE_INT_VERSION);

#ifdef _WIN32
        int main(int argc, const wchar_t* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
        int main(int argc, const wchar_t* const argv[], const
std::string& configFile, int version = ICE_INT_VERSION);
#endif

        int main(const StringSeq& args, const InitializationData&
initData = InitializationData(), int version = ICE_INT_VERSION);
        int main(const StringSeq& args, const std::string& configFile,
int version = ICE_INT_VERSION);

        virtual int run(int argc, char* argv[]) = 0;

        static const char* appName();

        static std::shared_ptr<Communicator> communicator();
        ...
    };
}
```

```
namespace Ice
{
    enum class SignalPolicy { HandleSignals, NoSignalHandling };

    class Application
    {
    public:

        Application(SignalPolicy = HandleSignals);
        virtual ~Application();

        int main(int argc, const char* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
        int main(int argc, const char* const argv[], const char*
configFile, int version = ICE_INT_VERSION);

#ifdef _WIN32
        int main(int argc, const wchar_t* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
        int main(int argc, const wchar_t* const argv[], const char*
configFile, int version = ICE_INT_VERSION);
#endif

        int main(const StringSeq& args, const InitializationData&
initData = InitializationData(), int version = ICE_INT_VERSION);
        int main(const StringSeq& args, const char* configFile, int
version = ICE_INT_VERSION);

        virtual int run(int argc, char* argv[]) = 0;

        static const char* appName();

        static CommunicatorPtr communicator();
        ...
    };
}
```

```
namespace Ice
{
    public abstract class Application
    {
        public abstract int run(string[] args);

        public Application();

        public Application(SignalPolicy signalPolicy);

        public int main(string[] args);
        public int main(string[] args, string configFile);
        public int main(string[] args, InitializationData init);

        public static string appName();

        public static Communicator communicator();

        ...
    }
}
```

```
package com.zeroc.Ice;

public enum SignalPolicy { HandleSignals, NoSignalHandling }

public abstract class Application
{
    public Application()

    public Application(SignalPolicy signalPolicy)

    public final int main(String appName, String[] args)

    public final int main(String appName, String[] args,
String configFile)

    public final int main(String appName, String[] args,
InitializationData initData)

    public abstract int run(String[] args)

    public static String appName()

    public static Communicator communicator()

    // ...
}
```

```
package Ice;

public enum SignalPolicy { HandleSignals, NoSignalHandling }

public abstract class Application
{
    public Application()

    public Application(SignalPolicy signalPolicy)

    public final int main(String appName, String[] args)

    public final int main(String appName, String[] args,
String configFile)

    public final int main(String appName, String[] args,
InitializationData initData)

    public abstract int run(String[] args)

    public static String appName()

    public static Communicator communicator()

    // ...
}
```

```
# in Ice module

class Application(object):

    def __init__(self, signalPolicy=0):

    def main(self, args, configFile=None, initData=None):

    def run(self, args):

    def appName():
        # ...
    appName = staticmethod(appName)

    def communicator():
        # ...
    communicator = staticmethod(communicator)
```

```
module Ice
    class Application
        def main(args, configFile=nil, initData=nil)

        def run(args)

        def Application.appName()

        def Application.communicator()
    end
end
```

The intent of this class is that you specialize `Application` and implement the pure virtual function or abstract method `run` in your derived class. Whatever code you would normally place in `main` goes into the `run` function/method instead. With Ice `Application`, the `main` of a typical Ice-based application becomes:

C++11 C++98 C# Java Java Compat Python Ruby

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[]) override
    {
        // application code goes here...

        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[])
    {
        // application code goes here...

        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

```
using System;

public class Server
{
    class App : Ice.Application
    {
        public override int run(string[] args)
        {
            // application code goes here...

            return 0;
        }
    }

    public static void Main(string[] args)
    {
        App app = new App();
        Environment.Exit(app.main(args));
    }
}
```

```
public class Server extends com.zeroc.Ice.Application
{
    public int run(String[] args)
    {
        // Application code goes here...

        return 0;
    }

    public static void main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

```
public class Server extends Ice.Application
{
    public int run(String[] args)
    {
        // Application code goes here...

        return 0;
    }

    public static void main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

```
import sys, Ice

class Server(Ice.Application):
    def run(self, args):
        # Application code goes here...
        return 0

app = Server()
status = app.main(sys.argv)
sys.exit(status)
```

```
require 'Ice'

class Client < Ice::Application
    def run(args)
        # Client code here...
        return 0
    end
end

app = Client.new()
status = app.main(ARGV)
exit(status)
```

Note that the main function or method of `Application` is overloaded: you can pass a string sequence instead of the main program's arguments. This is useful if you need to parse application-specific property settings on the command line. You also can call `main` with an optional file name or an `InitializationData` structure.

If you pass a configuration file name to `main`, the property settings in this file are overridden by settings in a file identified by the `ICE_CONFIG` environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The `Application main` function or method does the following:

1. It installs an exception handler for Ice exceptions. If your code fails to handle an Ice exception, `Application main` prints the exception details on `stderr` before returning with a non-zero return value.
2. In C++, it installs an exception handler for strings. This allows you to terminate your application in response to a fatal error condition by throwing a `std::string` or `const char*`. `Ice::Application` prints the string on `stderr` before returning a non-zero return value.
3. It initializes (by calling `Ice initialize`) and finalizes (by calling `Communicator::destroy`) a communicator. You can get access to the communicator for your application by calling the static `communicator()` member function or method.
4. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your `run` function or method is free of Ice-related options and only contains options and arguments that are specific to your application.
5. It sets the property `Ice.ProgramName` to the name of your application, and provides this name via the static `appName` member function or method.
6. It installs a signal handler that properly destroys the communicator; in Java, it installs a Shutdown Hook.
7. It installs a per-process logger if the application has not already configured one. The per-process logger uses the value of the `Ice.ProgramName` property as a prefix for its messages and sends its output to the standard error channel. An application can also specify an alternate logger.

Using `Application` ensures that your program properly finalizes the Ice run time, whether your application terminates normally or in response to an exception or signal.

Application and Threads
`Application` should be considered single-threaded until you call `main`. Then, in `run`, you can call concurrently `Application`'s functions or methods from multiple threads. If you create threads in `run`, you must join them before or when `run` returns.

The thread you use to call `main` is used to initialize the `Communicator` and then to call `run`. On Linux and macOS with C++, this thread must be the only thread of the process at the time you call `main` for signal handling to operate correctly.

Catching CTRL-C and Other Signals

The simple server we developed in Hello World Application had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is

particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, `Application` catches CTRL-C and similar signals, and allows you to select how to handle them.

C++11 C++98 C# Java Java Compat Python Ruby

Signals Caught
Linux and macOS: `SIGINT`, `SIGHUP`, `SIGTERM`

Windows: `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, `CTRL_SHUTDOWN_EVENT`

```
namespace Ice
{
    class Application
    {
    public:
        static void destroyOnInterrupt();
        static void shutdownOnInterrupt();
        static void ignoreInterrupt();
        static void callbackOnInterrupt();
        static void holdInterrupt();
        static void releaseInterrupt();
        static bool interrupted();

        virtual void interruptCallback(int signal);
    };
}
```

Signals Caught
Linux and macOS: `SIGINT`, `SIGHUP`, `SIGTERM`

Windows: `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, `CTRL_SHUTDOWN_EVENT`

```
namespace Ice
{
    class Application
    {
    public:
        static void destroyOnInterrupt();
        static void shutdownOnInterrupt();
        static void ignoreInterrupt();
        static void callbackOnInterrupt();
        static void holdInterrupt();
        static void releaseInterrupt();
        static bool interrupted();

        virtual void interruptCallback(int signal);
    };
}
```

```
namespace Ice
{
    public abstract class Application
    {
        // ...

        public static void destroyOnInterrupt();
        public static void shutdownOnInterrupt();
        public static void ignoreInterrupt();
        public static void callbackOnInterrupt();
        public static void holdInterrupt();
        public static void releaseInterrupt();

        public static bool interrupted();

        public virtual void interruptCallback(int sig);
    }
}
```

```
package com.zeroc.Ice;

public abstract class Application
{
    // ...

    synchronized public static void destroyOnInterrupt()
    synchronized public static void shutdownOnInterrupt()
    synchronized public static void setInterruptHook(Runnable r)
    synchronized public static void defaultInterrupt()
    synchronized public static boolean interrupted()
}
```

```
package Ice;

public abstract class Application
{
    // ...

    synchronized public static void destroyOnInterrupt()
    synchronized public static void shutdownOnInterrupt()
    synchronized public static void setInterruptHook(Runnable r)
    synchronized public static void defaultInterrupt()
    synchronized public static boolean interrupted()
}
```

```
class Application(object):
    # ...
    def destroyOnInterrupt():
        # ...
    destroyOnInterrupt = classmethod(destroyOnInterrupt)

    def shutdownOnInterrupt():
        # ...
    shutdownOnInterrupt = classmethod(shutdownOnInterrupt)

    def ignoreInterrupt():
        # ...
    ignoreInterrupt = classmethod(ignoreInterrupt)

    def callbackOnInterrupt():
        # ...
    callbackOnInterrupt = classmethod(callbackOnInterrupt)

    def holdInterrupt():
        # ...
    holdInterrupt = classmethod(holdInterrupt)

    def releaseInterrupt():
        # ...
    releaseInterrupt = classmethod(releaseInterrupt)

    def interrupted():
        # ...
    interrupted = classmethod(interrupted)

    def interruptCallback(self, sig):
        # Default implementation does nothing.
        pass
```

```
class Application
    def Application.destroyOnInterrupt()

    def Application.ignoreInterrupt()

    def Application.callbackOnInterrupt()

    def Application.holdInterrupt()

    def Application.releaseInterrupt()

    def Application.interrupted()

    def interruptCallback(sig):
        # Default implementation does nothing.
    end
    # ...
end
```

These functions or methods behave as follows:

Java Other

- `destroyOnInterrupt`
  This method installs a shutdown hook that calls `destroy` on the communicator. This is the default behavior.
- `shutdownOnInterrupt`
  This method installs a shutdown hook that calls `shutdown` on the communicator.
- `setInterruptHook`
  This method installs a custom shutdown hook (a `Runnable`) that takes responsibility for performing whatever action is necessary to terminate the application.
- `defaultInterrupt`
  This method removes the shutdown hook.
- `interrupted`
  This method returns true if the shutdown hook caused the communicator to shut down, false otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by the JVM. This is useful, for example, for logging purposes.
- `destroyOnInterrupt`
  This function or method registers a callback that destroys the communicator when one of the monitored signals is raised. This is the default behavior.
- `shutdownOnInterrupt`
  This function or method registers a callback that shuts down the communicator when one of the monitored signals is raised.
- `ignoreInterrupt`
  This function or method causes signals to be ignored.
- `callbackOnInterrupt`
  This function or method configures `Application` to invoke `interruptCallback` when a signal occurs, thereby giving the subclass responsibility for handling the signal.
- `holdInterrupt`
  This function or method causes signals to be held.

- `releaseInterrupt`
  This function or method restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.

- `interrupted`
  This function or method returns `true` if a signal caused the communicator to shut down, `false` otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.

- `interruptCallback`
  A subclass overrides this function or method to respond to signals. The Ice run time may call this function or method concurrently with any other thread. If it raises an exception, the Ice run time prints a warning on `stderr` and ignores the exception.

By default, `Application` behaves as if `destroyOnInterrupt` was invoked, therefore our `main` function requires no change to ensure that the program terminates cleanly on receipt of a signal.

> You can disable the signal-handling functionality of `Application` by passing the enumerator `NoSignalHandling` (or `1` in Python) to the constructor. In that case, signals retain their default behavior, that is, terminate the application.

Back to our application, we can add a diagnostic to report the occurrence of a signal:

C++11 C++98 C# Java Java Compat Python Ruby

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[]) override
    {
        // application code goes here...

        if(interrupted())
        {
            cerr << appName() << ": terminating" << endl;
        }
        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[])
    {
        // application code goes here...
        if(interrupted())
        {
            cerr << appName() << ": terminating" << endl;
        }
        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

```
using System;

public class Server
{
    class App : Ice.Application
    {
        public override int run(string[] args)
        {
            // Server code here...

            if(interrupted())
            {
                Console.Error.WriteLine(appName() + ": terminating");
            }

            return 0;
        }
    }

    public static void Main(string[] args)
    {
        App app = new App();
        Environment.Exit(app.main(args));
    }
}
```

```
public class Server extends Application
{
    public int run(String[] args)
    {
        // Application code goes here...

        if(interrupted())
        {
            System.err.println(appName() + ": terminating");
        }

        return 0;
    }

    public static void main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

```
public class Server extends Application
{
    public int run(String[] args)
    {
        // Application code goes here...

        if(interrupted())
        {
            System.err.println(appName() + ": terminating");
        }

        return 0;
    }

    public static void main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

```
import sys, Ice

class MyApplication(Ice.Application):
    def run(self, args):

        # Application code goes here...

        if self.interrupted():
            print self.appName() + ": terminating"

        return 0

app = MyApplication()
status = app.main(sys.argv)
sys.exit(status)
```

```
require 'Ice'

class MyApplication < Ice::Application
    def run(args)
        # Application code goes here...

        if Ice::Application::interrupted()
            print Ice::Application::appName() + ": terminating"
        end

        return 0
    end
end

app = MyApplication.new()
status = app.main(ARGV)
exit(status)
```

Note that, if your server is interrupted by a signal, the Ice run time waits for all currently executing operation dispatches to finish. This means that an operation that updates persistent state cannot be interrupted in the middle of what it was doing and cause partial update problems.

C++ Applications on Linux and macOS
If you handle signals with your own handler (by deriving a subclass from `Ice::Application` and calling `callbackOnInterrupt`), the handler is invoked synchronously from a separate thread. This means that the handler can safely call into the Ice run time or make system calls that are not async-signal-safe without fear of deadlock or data corruption. Note that `Ice::Application` blocks delivery of `SIGINT`, `SIGHUP`, and `SIGTERM`. If your application calls `exec`, this means that the child process will also ignore these signals; if you need the default behavior of these signals in the `exec`'d process, you must explicitly reset them to `SIG_DFL` before calling `exec`.

Java Shutdown Hook and Main Thread
In a subclass of `Application`, the default shutdown hook (as installed by `destroyOnInterrupt`) blocks until the application's main thread completes. As a result, an interrupted application may not terminate successfully if the main thread is blocked. For

example, this can occur in an interactive application when the main thread is waiting for console input. To remedy this situation, the application can install an alternate shutdown hook that does not wait for the main thread to finish:

Java

```
public class Server extends com.zeroc.Ice.Application
{
    public int run(String[] args)
    {
        setInterruptHook(() -> { communicator.destroy(); });

        // ...
    }
}
```

After replacing the default shutdown hook using `setInterruptHook`, the JVM will terminate as soon as the communicator is destroyed.

Limitations of `Application`

`Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use Ice `Application`. Instead, you must use create the `Communicators` directly with `initialize` or through `CommunicatorHolders` in C++, as we saw in the Hello World Application.

See Also

- Communicator Initialization
- Command-Line Parsing and Initialization
- CtrlCHandler Helper Class

CtrlCHandler Helper Class

The C++ class `CtrlCHandler` provides a portable abstraction to handle CTRL-C and similar signals sent to a C++ process. On Windows, `CtrlCHandler` is a wrapper for `SetConsoleCtrlHandler`; on POSIX platforms, it handles `SIGHUP`, `SIGTERM` and `SIGINT` with a dedicated thread that waits for these signals using `sigwait`. Signals are handled by a callback function that you implement and register. The callback is a simple function that takes an `int` (the signal number) and returns `void`; it must not throw any exception.

C++11 C++98

```
namespace Ice
{
    class CtrlCHandler
    {
    public:
        explicit CtrlCHandler(std::function<void(int)> = nullptr);
        ~CtrlCHandler();

        std::function<void(int)> setCallback(std::function<void(int>));
        std::function<void(int)> getCallback() const;
    };
}
```

```
namespace Ice
{
    typedef void (*CtrlCHandlerCallback)(int);

    class CtrlCHandler
    {
    public:
        explicit CtrlCHandler(CtrlCHandlerCallback = 0);
        ~CtrlCHandler();

        CtrlCHandlerCallback setCallback(CtrlCHandlerCallback);
        CtrlCHandlerCallback getCallback() const;
    };
}
```

The `CtrlCHandler` does not terminate your application after running the registered callback (if any). `CtrlCHandler` allows you handle CTRL-C and similar signals repeatedly.

The member functions of `CtrlCHandler` behave as follows:

- `CtrlCHandler`
  Constructs an instance with optionally a callback function. Only one instance of `CtrlCHandler` can exist in a process at a given moment in time. On POSIX platforms, the constructor masks `SIGHUP`, `SIGTERM` and `SIGINT`, then starts a thread that waits for these signals using `sigwait`. For signal masking to work properly, **it is imperative that the `CtrlCHandler` instance be created before starting any thread**, and in particular before initializing an Ice communicator.
- `~CtrlCHandler`
  Destroys the instance, after which the default signal processing behavior is restored on Windows (`TerminateProcess`). On POSIX

platforms, the "sigwait" thread is terminated and joined, but the signal mask remains unchanged, so subsequent signals are ignored.

- `setCallback`
  Sets a new callback function, and returns the old callback function.
- `getCallback`
  Gets the current callback function.

It is legal specify `nullptr` for the callback function, in which case signals are caught and ignored until a non-null callback function is set.

A typical use for `CtrlCHandler` is to shutdown a communicator in an Ice server. For example, you could use it as follows:

C++11 C++98

```
int
main(int argc, char* argv[])
{
    Ice::CtrlCHandler ctrlCHandler;
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);
        ctrlCHandler.setCallback([communicator = ich.communicator()](int
signal) // C++14 syntax
                                 {
                                     cerr << "caught signal " << signal
<< ", shutting down communicator" << endl;
                                     communicator->shutdown(); //
shutdown is noexcept
                                 });
        auto adapter =
            ich->createObjectAdapterWithEndpoints("Hello", "default -h
localhost -p 10000");
        adapter->add(make_shared<HelloI>(),
Ice::stringToIdentity("hello"));
        adapter->activate();
        ich->waitForShutdown();
    }
    catch(const std::exception& ex)
    {
        cerr << ex.what() << endl;
        return 1;
    }
    return 0;
}
```

```
Ice::CommunicatorPtr communicator;
void shutdownCommunicator(int);

int
main(int argc, char* argv[])
{
    Ice::CtrlCHandler ctrlCHandler;
    try
    {
        Ice::CommunicatorHolder ich(argc, argv);
        communicator = ich.communicator();
        ctrlCHandler.setCallback(shutdownCommunicator);

        Ice::ObjectAdapterPtr adapter =
            ich->createObjectAdapterWithEndpoints("Hello", "default -h
localhost -p 10000");
        adapter->add(new HelloI, Ice::stringToIdentity("hello"));
        adapter->activate();
        ich->waitForShutdown();
    }
    catch(const std::exception& ex)
    {
        cerr << ex.what() << endl;
        return 1;
    }
    return 0;
}

void
shutdownCommunicator(int signal)
{
    cerr << "caught signal " << signal << ", shutting down communicator"
<< endl;
    communicator->shutdown(); // shutdown does not throw any exception
}
```

Service Helper Class

On this page:

The `Ice::Service` C++ Class

Ice provides `Ice::Service`, a singleton class that is comparable to `Ice::Application` but also encapsulates the low-level, platform-specific initialization and shutdown procedures common to Unix daemons and Windows services. `Ice::Service` is currently available only in C++.

The `Ice::Service` class is declared as follows:

C++11 C++98

```
namespace Ice
{
    class Service
    {
    public:

        Service();
        virtual ~Service();

        virtual bool shutdown();
        virtual void interrupt();

        int main(int argc, const char* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
#ifdef _WIN32
        int main(int argv, const wchar_t* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
#endif
        int main(const StringSeq& args, const InitializationData&
initData = InitializationData(), int version = ICE_INT_VERSION);

        std::shared_ptr<Communicator> communicator() const;
        static Service* instance();

        bool service() const;
        std::string name() const;
        bool checkSystem() const;

        int run(int argc, const char* const argv[], const
InitializationData& initData = InitializationData(), int version=
ICE_INT_VERSION);
#ifdef _WIN32
```

```
        int run(int argc, const wchar_t* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
#endif

#ifdef _WIN32
        void configureService(const std::string& name);
#else
        void configureDaemon(bool chdir, bool close, const std::string&
pidFile);
#endif

        virtual void handleInterrupt(int);

    protected:

        virtual bool start(int argc, char* argv[], int& status) = 0;
        virtual void waitForShutdown();
        virtual bool stop();

        virtual std::shared_ptr<Communicator>
initializeCommunicator(int& argc, char* argv[], const
InitializationData& initData, int version);

        virtual void syserror(const std::string& msg);
        virtual void error(const std::string& msg);
        virtual void warning(const std::string& msg);
        virtual void trace(const std::string& msg);
        virtual void print(const std::string& msg);

        void enableInterrupt();
        void disableInterrupt();
```

```
        ...
    };
}
```

```
namespace Ice
{
    class Service
    {
    public:

        Service();
        virtual ~Service();

        virtual bool shutdown();
        virtual void interrupt();

        int main(int argc, const char* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
#ifdef _WIN32
        int main(int argv, const wchar_t* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
#endif
        int main(const StringSeq& args, const InitializationData&
initData = InitializationData(), int version = ICE_INT_VERSION);

        CommunicatorPtr communicator() const;
        static Service* instance();

        bool service() const;
        std::string name() const;
        bool checkSystem() const;

        int run(int argc, const char* const argv[], const
InitializationData& initData = InitializationData(), int version=
ICE_INT_VERSION);
#ifdef _WIN32
        int run(int argc, const wchar_t* const argv[], const
InitializationData& initData = InitializationData(), int version =
ICE_INT_VERSION);
#endif

#ifdef _WIN32
        void configureService(const std::string& name);
#else
        void configureDaemon(bool chdir, bool close, const std::string&
```

```
pidFile);
#endif

        virtual void handleInterrupt(int);

    protected:

        virtual bool start(int argc, char* argv[], int& status) = 0;
        virtual void waitForShutdown();
        virtual bool stop();

        virtual CommunicatorPtr initializeCommunicator(int& argc, char*
argv[], const InitializationData& initData, int version);

        virtual void syserror(const std::string& msg);
        virtual void error(const std::string& msg);
        virtual void warning(const std::string& msg);
        virtual void trace(const std::string& msg);
        virtual void print(const std::string& msg);

        void enableInterrupt();
        void disableInterrupt();
```

```
        ...
    };
}
```

At a minimum, an Ice application that uses the `Ice::Service` class must define a subclass and override the `start` member function, which is where the service must perform its startup activities, such as processing command-line arguments, creating an object adapter, and registering servants. The application's `main` function must instantiate the subclass and typically invokes its `main` member function, passing the program's argument vector as parameters. The example below illustrates a minimal `Ice::Service` subclass:

C++11 C++98

```
#include <Ice/Ice.h>

class MyService : public Ice::Service
{
protected:
    virtual bool start(int, char*[], int&) override;
private:
    std::shared_ptr<Ice::ObjectAdapter> _adapter;
};

bool
MyService::start(int argc, char* argv[], int& status)
{
    _adapter = communicator()->createObjectAdapter("MyAdapter");
    _adapter->addWithUUID(std::make_shared<MyServantI>());
    _adapter->activate();
    status = EXIT_SUCCESS;
    return true;
}

int
main(int argc, char* argv[])
{
    MyService svc;
    return svc.main(argc, argv);
}
```

```
#include <Ice/Ice.h>

class MyService : public Ice::Service
{
protected:
    virtual bool start(int, char*[], int&);
private:
    Ice::ObjectAdapterPtr _adapter;
};

bool
MyService::start(int argc, char* argv[], int& status)
{
    _adapter = communicator()->createObjectAdapter("MyAdapter");
    _adapter->addWithUUID(new MyServantI);
    _adapter->activate();
    status = EXIT_SUCCESS;
    return true;
}

int
main(int argc, char* argv[])
{
    MyService svc;
    return svc.main(argc, argv);
}
```

The `Service::main` member function performs the following sequence of tasks:

1. Scans a copy of the argument vector for reserved options that indicate whether the program should run as a system service and removes these options from this copy of the argument vector. Additional reserved options are supported for administrative tasks.
2. Configures the program for running as a system service (if necessary) by invoking `configureService` or `configureDaemon`, as appropriate for the platform.
3. Invokes the `run` member function with the filtered argument vector and returns its result.

Note that, as for `Application::main`, `Service::main` is overloaded to accept a string sequence instead of an `argc`/`argv` pair. This is useful if you need to parse application-specific property settings on the command line.

> For maximum portability, we strongly recommend that all initialization tasks be performed in the `start` member function and not in the global `main` function. For example, allocating resources in `main` can cause program instability for Unix daemons.

The `Service::run` member function executes the service in the steps shown below:

1. Makes a copy of the provided argument vector.
2. Installs a signal handler.
3. Invokes the `initializeCommunicator` member function to obtain a communicator. The communicator instance can be accessed using the `communicator` member function.
4. Invokes the `start` member function. If `start` returns `false` to indicate failure, `run` destroys the communicator and returns immediately using the exit status provided in `status`.
5. Invokes the `waitForShutdown` member function, which should block until `shutdown` is invoked.
6. Invokes the `stop` member function. If `stop` returns `true`, `run` considers the application to have terminated successfully.
7. Destroys the communicator.
8. Gracefully terminates the system service (if necessary).

If an unhandled exception is caught by `Service::run`, a descriptive message is logged, the communicator is destroyed and the service is terminated.

`Ice::Service` Member Functions

The virtual member functions in `Ice::Service` represent the points at which a subclass can intercept the service activities. All of the virtual member functions (except `start`) have default implementations.

- `void handleInterrupt(int sig)`
  Invoked by the signal handler when it catches a signal. The default implementation ignores the signal if it represents a logoff event and the `Ice.Nohup` property is set to a value larger than zero, otherwise it invokes the `interrupt` member function.

- `std::shared_ptr<Ice::Communicator> initializeCommunicator(int& argc, char* argv[], const InitializationData& initData, int version)` (C++11)
  `Ice::CommunicatorPtr initializeCommunicator(int& argc, char* argv[], const InitializationData& initData, int version)` (C++98)
  Initializes a communicator. The default implementation invokes `Ice::initialize` and passes the given arguments.

- `void interrupt()`
  Invoked by the signal handler to indicate a signal was received. The default implementation invokes the `shutdown` member function.

- `bool shutdown()`
  Causes the service to begin the shutdown process. The default implementation invokes `shutdown` on the communicator. The subclass must return `true` if shutdown was started successfully, and `false` otherwise.

- `bool start(int argc, char* argv[], int& status)`
  Allows the subclass to perform its startup activities, such as scanning the provided argument vector for recognized command-line options, creating an object adapter, and registering servants. The subclass must return `true` if startup was successful, and `false` o therwise. The subclass can set an exit status via the `status` parameter. This status is returned by `main`.

- `bool stop()`
  Allows the subclass to clean up prior to termination. The default implementation does nothing but return `true`. The subclass must return `true` if the service has stopped successfully, and `false` otherwise.

- `void syserror(const std::string& msg)`
  `void error(const std::string& msg)`
  `void warning(const std::string& msg)`
  `void trace(const std::string& msg)`
  `void print(const std::string& msg)`
  Convenience functions for logging messages to the communicator's logger. The `syserror` member function includes a description of the system's current error code. You can also log messages to these functions using utility classes similar to the C++ Logger Utility Classes: these classes are `ServiceSysError`, `ServiceError`, `ServiceWarning`, `ServiceTrace` and `ServicePrint`, all nested in the Service class.

- `void waitForShutdown()`
  Waits indefinitely for the service to shut down. The default implementation invokes `waitForShutdown` on the communicator.

The non-virtual member functions shown in the class definition are described below:

- `bool checkSystem() const`
  Returns true if the operating system supports Windows services or Unix daemons.

- `std::shared_ptr<Ice::Communicator> communicator() const` (C++11)
  `Ice::CommunicatorPtr communicator() const` (C++98)
  Returns the communicator used by the service, as created by `initializeCommunicator`.

- `void configureDaemon(bool chdir, bool close, const std::string& pidFile)`
  Configures the program to run as a Unix daemon. The `chdir` parameter determines whether the daemon changes its working directory to the root directory. The `close` parameter determines whether the daemon closes unnecessary file descriptors (i.e., stdin, stdout, etc.). If a non-empty string is provided in the `pidFile` parameter, the daemon writes its process ID to the given file.

- `void configureService(const std::string& name)`
  Configures the program to run as a Windows service with the given name.

- `void disableInterrupt()`
  Disables the signal handling behavior in `Ice::Service`. When disabled, signals are ignored.

- `void enableInterrupt()`
  Enables the signal handling behavior in `Ice::Service`. When enabled, the occurrence of a signal causes the `handleInterrupt` member function to be invoked.

- `static Service* instance()`
  Returns the singleton `Ice::Service` instance.

- `int main(int argv, const char* const argv[], const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION)`
  `int main(int argc, const wchar_t* const argv[], const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION)`
  `int main(const Ice::StringSeq& args, const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION);`
  The primary entry point of the `Ice::Service` class. The tasks performed by this function are described earlier in this section. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure. For Windows, this function is overloaded to allow you to pass a `wchar_t` argument vector.

- `std::string name() const`
  Returns the name of the service. If the program is running as a Windows service, the return value is the Windows service name, otherwise it returns the value of `argv[0]`.

- `int run(int argc, const char* const argv[], const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION)`
  Alternative entry point for applications that prefer a different style of service configuration. The program must invoke `configureService` (Windows) or `configureDaemon` (Unix) in order to run as a service. The tasks performed by this function were described earlier. The function normally returns `EXIT_SUCCESS` or `EXIT_FAILURE`, but the `start` method can also supply a different value via its `status` argument.

- `bool service() const`
  Returns true if the program is running as a Windows service or Unix daemon, or false otherwise.

Unix Daemons

On Linux and macOS, passing `--daemon` causes your program to run as a daemon. When this option is present, `Ice::Service` performs the following additional actions:

- Creates a background child process in which `Service::main` performs its tasks. The foreground process does not terminate until the child process has successfully invoked the `start` member function. This behavior avoids the uncertainty often associated with starting a daemon from a shell script by ensuring that the command invocation does not complete until the daemon is ready to receive requests.
- Changes the current working directory of the child process to the root directory, unless `--nochdir` is specified.
- Closes all file descriptors, unless `--noclose` is specified. The standard input (stdin) channel is closed and reopened to `/dev/null`. Likewise, the standard output (stdout) and standard error (stderr) channels are also closed and reopened to `/dev/null` unless `Ice.StdOut` or `Ice.StdErr` are defined, respectively, in which case those channels use the designated log files.

  > The file descriptors are not closed until after the communicator is initialized, meaning standard input, standard output, and standard error are available for use during this time. For example, the IceSSL plug-in may need to prompt for a passphrase on standard input, or Ice may print the child's process id on standard output if the property `Ice.PrintProcessId` is set.

The following additional command-line options can be specified in conjunction with the `--daemon` option:

- `--pidfile` *`FILE`*
  This option writes the process ID of the service into the specified *`FILE`*.

- `--noclose`
  Prevents `Ice::Service` from closing unnecessary file descriptors. This can be useful during debugging and diagnosis because it provides access to the output from the daemon's standard output and standard error.

- `--nochdir`
  Prevents `Ice::Service` from changing the current working directory.

All of these options are removed from the argument vector that is passed to the `start` member function.

We strongly recommend that you perform all initialization tasks in your service's `start` member function, and not in the global `main` function. This is especially important for process-specific resources such as file descriptors, threads, and mutexes, which can be affected by the use of the `fork` system call in `Ice::Service`. For example, any files opened in `main` are automatically closed by `Ice::Service` and therefore unusable in your service, unless the daemon is started with the `--noclose` option.

Windows Services

On Windows, `Ice::Service` recognizes the following command-line options:

- `--service` *NAME*
  Run as a Windows service named *NAME*, which must already be installed. This option is removed from the argument vector that is passed to the `start` member function.

Installing and configuring a Windows service is outside the scope of the `Ice::Service` class. Ice includes a utility for installing its services which you can use as a model for your own applications.

The `Ice::Service` class supports the Windows service control codes `SERVICE_CONTROL_INTERROGATE` and `SERVICE_CONTROL_STOP`. Upon receipt of `SERVICE_CONTROL_STOP`, `Ice::Service` invokes the `shutdown` member function.

`Ice::Service` Logging Considerations

A service that uses a custom logger has several ways of configuring it:

- as a process-wide logger,
- in the `InitializationData` argument that is passed to `main`,
- by overriding the `initializeCommunicator` member function.

On Windows, `Ice::Service` installs its own logger that uses the Windows `Application` event log if no custom logger is defined. The source name for the event log is the service's name unless a different value is specified using the property `Ice.EventLog.Source`.

On Unix, the default Ice logger (which logs to the standard error output) is used when no other logger is configured. For daemons, this is not appropriate because the output will be lost. To change this, you can either implement a custom logger or set the `Ice.UseSyslog` property, which selects a logger implementation that logs to the `syslog` facility. Alternatively, you can set the `Ice.LogFile` property to write log messages to a file.

Note that `Ice::Service` may encounter errors before the communicator is initialized. In this situation, `Ice::Service` uses its default logger unless a process-wide logger is configured. Therefore, even if a failing service is configured to use a different logger implementation, you may find useful diagnostic information in the `Application` event log (on Windows) or sent to standard error (on Linux and macOS).

See Also

- Hello World Application
- Properties and Configuration
- Communicator Initialization
- Logger Facility
- Windows Services

Object Identity

On this page:

The `Ice::Identity` Type

Each Ice object has an object identity defined as follows:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
}
```

As you can see, an object identity consists of a pair of strings, a `name` and a `category`. The complete object identity is the combination of `name` and `category`, that is, for two identities to be equal, both `name` and `category` must be the same. The `category` member is usually the empty string, unless you are using servant locators, default servants or callbacks with Glacier2.

If `name` is an empty string, `category` must be the empty string as well. (An identity with an empty `name` and a non-empty `category` is illegal.) If a proxy contains an identity in which `name` is empty, Ice interprets that proxy as a null proxy.

Object identities can be represented as strings; the category part appears first and is followed by the name; the two components are separated by a `/` character, for example:

```
Factory/File
```

In this example, `Factory` is the category, and `File` is the name. If the `name` or `category` member themselves contain a `/` character, the stringified representation escapes the `/` character with a `\`, for example:

```
Factories\/Factory/Node\/File
```

In this example, the category is `Factories/Factory` and the name is `Node/File`.

Syntax for Stringified Identities

You rarely need to write identities as strings because, typically, your code will be using the identity helper functions `identityToString` and `stringToIdentity`, or simply deal with proxies instead of identities. However, on occasion, you will need to use stringified identities in configuration files. If the identities happen to contain meta-characters (such as a slash or backslash), or characters outside the printable ASCII range, these characters may need to be escaped in the stringified representation.

Here are rules that the Ice run time applies when parsing a stringified identity:

1. The parser scans the stringified identity for an unescaped slash character (`/`). If such a slash character can be found, the substrings to the left and right of the slash are parsed as the `category` and `name` members of the identity, respectively; if no such slash character can be found, the entire string is parsed as the `name` member of the identity, and the `category` member is the empty string.
2. Each of the `category` (if present) and `name` substrings are parsed like Slice String Literals, except that an escaped slash character (`\/`) is converted into a simple slash (`/`).

Identity Helper Functions

To make conversion of identities to and from strings easier, Ice provides functions to convert an Identity to and from a native string, using the string format described in the preceding paragraph. These helper functions are called `identityToString` (to stringify an identity into a string) and `stringToIdentity` (to parse a stringified identity and create the corresponding identity).

`ToStringMode` Enumeration

When *stringifying* an identity with `identityToString`, you can choose the algorithm, or mode, used in this "to string" implementation. These modes correspond to the enumerators of the `Ice::ToStringMode` enumeration:

Slice

```
module Ice
{
    local enum ToStringMode { Unicode, ASCII, Compat }
}
```

These modes are used only when you create a stringified identity or proxy. The resulting strings are all in the same format.

The selected mode affects only the handling of non-ASCII characters and non-printable ASCII characters, such as the ASCII character with ordinal value 127 (delete).

The table below shows how these characters are encoded depending of the selected `ToStringMode`:

ToStringMode	Non-Printable ASCII Character other than \a, \b, \f, \n, \r, \t and \v	Non-ASCII Character	Notes
`Unicode`	Replaced by its short universal character name. For example \u007F (delete).	Not escaped - kept as is.	The resulting string contains printable ASCII characters plus possibly non-ASCII characters.
`ASCII`	Replaced by its short universal character name, just like `Unicode`.	If the character's code point is in the BMP, it's escaped with its short universal character name (\unnnn); if the character is outside the BMP, it's escaped with its long universal character name (\Unnnnnnnn). For example \u20ac (euro sign) and \U0001F34C (banana symbol).	The resulting string contains only printable ASCII characters.
`Compat`	Replaced by an octal escape sequence with 3 digits, \ooo. For example \177 (delete).	Replaced by a sequence of 3-digit octal escape sequences, that represent the UTF-8 encoding of this character. For example \342\202\254 (euro sign) and \360\237\215\214 (banana symbol)	The resulting string contains only printable ASCII characters.

The default mode is `Unicode`.

The `Compat` mode is provided for backwards-compatibility with Ice 3.6 and earlier. These older versions do not recognize universal character names and reject non-printable ASCII characters in stringified identities.

Communicator::identityToString

The local interface Communicator provides an `identityToString` helper:

Slice

```
module Ice
{
    local interface Communicator
    {
        string identityToString(Identity id);
        // ...
    }
}
```

`identityToString` converts an identity to a string. The `Ice.ToStringMode` property of the communicator controls how non-printable ASCII characters and non-ASCII characters are represented in the resulting string.

identityToString and stringToIdentity

These language-native helper functions are defined as follows:

C++11 C++98 C# Java Java Compat JavaScript MATLAB ObjC PHP Python Ruby

```
namespace Ice
{
    std::string identityToString(const Identity&, ToStringMode =
ToStringMode::Unicode);
    Identity stringToIdentity(const std::string&);
}
```

```
namespace Ice
{
    std::string identityToString(const Identity&, ToStringMode =
Unicode);
    Identity stringToIdentity(const std::string&);
}
```

```
namespace Ice
{
    public sealed class Util
    {
        public static string identityToString(Identity id, ToStringMode
toStringMode = ToStringMode.Unicode);
        public static Identity stringToIdentity(string s);
    }
}
```

```
package com.zeroc.Ice;

public final class Util
{
    public static String identityToString(Identity id, ToStringMode
toStringMode);
    public static String identityToString(Identity id); // calls
identityToString with ToStringMode.Unicode
    public static Identity stringToIdentity(String s);
}
```

```
package Ice;

public final class Util
{
    public static String identityToString(Identity id, ToStringMode
toStringMode);
    public static String identityToString(Identity id); // calls
identityToString with ToStringMode.Unicode
    public static Identity stringToIdentity(String s);
}
```

```
Ice.stringToIdentity = function(s)
Ice.identityToString = function(ident, toStringMode =
Ice.ToStringMode.Unicode)
```

```
% in Ice package

function identity = stringToIdentity(s)
function s = identityToString(identity, toStringMode)
```

```
@interface ICEUtil : NSObject
+(NSMutableString*) identityToString:(ICEIdentity*)ident
toStringMode:(ICEToStringMode)toStringMode;
+(NSMutableString*) identityToString:(ICEIdentity*)ident; // calls
identityToString with ICEUnicode
+(NSMutableString*) identityToString:(ICEIdentity*)ident;
@end
```

```
namespace Ice
{
    function identityToString($ident, $toStringMode=null) // null
corresponds to the Unicode mode
    function stringToIdentity($str)
}
```

```
# in Ice module

def identityToString(ident, toStringMode=None) # None corresponds to the
Unicode mode
def stringToIdentity(str)
```

```
module Ice
    def Ice.identityToString(str, toStringMode=nil) # nil corresponds to
the Unicode mode
    def Ice.stringToIdentity(str)
```

See Also

- Servant Activation and Deactivation
- Servant Locators
- Default Servants
- Glacier2

Plug-in Facility

Ice provides a plug-in facility that allows you to load new features into your application, without changing this application's code. Depending on the programming language, a plug-in is packaged as a shared library or a static library, a set of Java classes, or a .NET assembly.

You control the loading and initialization of plug-ins with Ice configuration properties, and the plug-ins themselves are configured through Ice configuration properties.

Ice relies on plug-ins to implement a number of features, such as IceSSL (a secure transport for Ice over SSL/TLS).

This section describes the plug-in facility in more detail and demonstrates how to implement an Ice plug-in.

Topics

See Also

Plug-in API

On this page:

The `Plugin` Interface

The plug-in facility defines a local Slice interface that all plug-ins must implement:

Slice

```
module Ice
{
    local interface Plugin
    {
        void initialize();
        void destroy();
    }
}
```

The lifecycle of an Ice plug-in is structured to accommodate dependencies between plug-ins, such as when a logger plug-in needs to use IceSSL for its logging activities. Consequently, a plug-in object's lifecycle consists of four phases:

- Construction
  The Ice run time uses a language-specific factory API for instantiating plug-ins. During construction, a plug-in can acquire resources but must not spawn new threads or perform activities that depend on other plug-ins.
- Initialization
  After all plug-ins have been constructed, the Ice run time invokes `initialize` on each plug-in. The order in which plug-ins are initialized is undefined by default but can be customized using a configuration property. If a plug-in has a dependency on another plug-in, you must configure the Ice run time so that initialization occurs in the proper order. In this phase it is safe for a plug-in to spawn new threads; it is also safe for a plug-in to interact with other plug-ins and use their services, as long as those plug-ins have already been initialized. If `initialize` raises an exception, the Ice run time invokes `destroy` on all plug-ins that were successfully initialized (in the reverse order of initialization) and raises the original exception to the application.
- Active
  The active phase spans the time between initialization and destruction. Plug-ins must be designed to operate safely in the context of multiple threads.
- Destruction
  The Ice run time invokes `destroy` on each plug-in in the reverse order of initialization.

  > A plug-in's `destroy` implementation must not make remote invocations.

This lifecycle is repeated for each new communicator that an application creates and destroys.

C++ Plug-in Factory

In C++, a plug-in factory is a function with C linkage and the following signature:

C++11 C++98

C++

```
extern "C"
{
    Ice::Plugin* functionName(const std::shared_ptr<Ice::Communicator>&
communicator,
                              const std::string& name,
                              const Ice::StringSeq& args)
    {
       ...
    }
}
```

C++

```
extern "C"
{
    Ice::Plugin* functionName(const Ice::CommunicatorPtr& communicator,
                              const std::string& name,
                              const Ice::StringSeq& args)
    {
       ...
    }
}
```

You can choose any name for the factory function.

Since the function uses C linkage, it must return the plug-in object as a regular C++ pointer and not as a smart pointer. Furthermore, the function must not raise C++ exceptions; if an error occurs, the function must return zero. The arguments to the function consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

If your plug-in and the associated factory function are packaged in a shared library or DLL loaded at run time, you need to export this function from the shared library or DLL. We provide the macro `ICE_DECLSPEC_EXPORT` for this purpose:

C++

```
#if defined(_MSC_VER)
#   define ICE_DECLSPEC_EXPORT __declspec(dllexport)
...
#elif defined(__GNUC__) || defined(__clang__)
#   define ICE_DECLSPEC_EXPORT __attribute__((visibility ("default")))
...
```

Simply add `ICE_DECLSPEC_EXPORT` to the definition of your plug-in factory:

C++11 C++98

C++

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
functionName(const std::shared_ptr<Ice::Communicator>& communicator,

const std::string& name,

const Ice::StringSeq& args)
    {
       ...
    }
}
```

C++

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
functionName(const Ice::CommunicatorPtr& communicator,

const std::string& name,

const Ice::StringSeq& args)
    {
       ...
    }
}
```

If you don't want to rely on the dynamic loading of your plug-in shared library or DLL at run time, or if your plug-in is packaged in a static library, you can also link the plug-in into your application and call `Ice::registerPluginFactory` in your main application's code to regist er the plug-in before you initialize Ice communicators. For example:

C++

```
MyApp::MyApp()
{
   // Load/link the "IceSSL" plug-in before communicator initialization

   Ice::registerPluginFactory("IceSSL", createIceSSL, false);
}
```

The `registerPluginFactory` function registers the plug-in's factory function with the Ice run time. It returns `void`, and accepts the following parameters:

- `const string&`
  The name of the plug-in.

- `PLUGIN_FACTORY`
  A pointer to the plug-in factory function.
- `bool`
  When `true`, the plug-in is always loaded (created) during communicator initialization, even if `Ice.Plugin.`*`name`* is not set. When `false`, the plug-in is loaded (created) during communication initialization only if `Ice.Plugin.`*`name`* is set to a non-empty value (e.g.: `Ice.Plugin.IceSSL=1`).

Java Plug-in Factory

In Java, a plug-in factory must implement the `PluginFactory` interface:

Java Java Compat

```
package com.zeroc.Ice;

public interface PluginFactory
{
    Plugin create(Communicator communicator, String name,
String[] args);
}
```

```
package Ice;

public interface PluginFactory
{
    Plugin create(Communicator communicator, String name,
String[] args);
}
```

The arguments to the `create` method consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

The `create` method can return `null` to indicate that a general error occurred, or it can raise `PluginInitializationException` to provide more detailed information. If any other exception is raised, the Ice run time wraps it inside an instance of `PluginInitialization Exception`.

C# Plug-in Factory

In .NET, a plug-in factory must implement the `Ice.PluginFactory` interface:

C#

```
namespace Ice
{
    public interface PluginFactory
    {
        Plugin create(Communicator communicator, string name,
string[] args);
    }
}
```

The arguments to the `create` method consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

The `create` method can return `null` to indicate that a general error occurred, or it can raise `PluginInitializationException` to provide more detailed information. If any other exception is raised, the Ice run time wraps it inside an instance of `PluginInitialization Exception`.

See Also

- Plug-in Configuration
- Advanced Plug-in Topics

Plug-in Configuration

Plug-ins are installed using a configuration property of the following form:

```
Ice.Plugin.Name=entry_point [arg ...]
```

In most cases you can assign an arbitrary name to a plug-in. In the case of IceSSL, however, the plug-in requires that its name be `IceSSL`.

The value of *`entry_point`* is a language-specific representation of the plug-in's factory. In C++, it consists of the path name of the shared library or DLL containing the factory function, along with the name of the factory function. In Java and .NET, the entry point specifies the factory class.

The language-specific nature of plug-in properties can present a problem when applications that are written in multiple implementation languages attempt to share a configuration file. Ice supports an alternate syntax for plug-in properties that alleviates this issue:

```
Ice.Plugin.name.cpp=...    # C++ plug-in
Ice.Plugin.name.java=...   # Java plug-in
Ice.Plugin.name.clr=...    # .NET (Common Language Runtime) plug-in
```

Plug-in properties having a suffix of `.cpp`, `.java`, or `.clr` are loaded only by the appropriate Ice run time and ignored by others.

After extracting the plug-in's entry point from the property value, any remaining text is parsed using semantics similar to that of command-line arguments. Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes:

```
Ice.Plugin.MyPlugin=entry_point --load "C:\Data Files\config.dat"
```

Ice passes these arguments to the plug-in's entry point during loading.

See Also

- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder
- IceSSL

Advanced Plug-in Topics

This page discusses additional aspects of the Ice plug-in facility that may be of use to applications with special requirements.

On this page:

Plug-in Dependencies

If a plug-in has a dependency on another plug-in, you must ensure that Ice initializes the plug-ins in the proper order. Suppose that a custom plug-in depends on IceSSL; for example, it may need to make secure invocations on another server. We start with the following C++ configuration:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.MyPlugin=MyPlugin:createMyPlugin
```

The problem with this configuration is that it does not specify the order in which the plug-ins should be loaded and initialized. If the Ice run time happens to initialize `MyPlugin` first, the plug-in's `initialize` method will fail if it attempts to use the services of the uninitialized IceSSL plug-in.

To remedy the situation, we need to add one more property:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.MyPlugin=MyPlugin:createMyPlugin
Ice.PluginLoadOrder=IceSSL, MyPlugin
```

Using the `Ice.PluginLoadOrder` property we can guarantee that the plug-ins are loaded in the correct order.

> Plug-ins added manually via the plug-in manager are appended to the end of the plug-in list, in order of addition. The last plug-in added is the first to be destroyed.

The Plug-in Manager

`PluginManager` is the name of an internal Ice object that is responsible for managing all aspects of Ice plug-ins. This object supports a local Slice interface of the same name, and an application can obtain a reference to this object using the following communicator operation:

Slice

```
module Ice
{
    local interface Communicator
    {
        PluginManager getPluginManager();
        // ...
    }
}
```

The `PluginManager` interface offers three operations:

Slice

```
module Ice
{
    local interface PluginManager
    {
        void initializePlugins();
        Plugin getPlugin(string name);
        void addPlugin(string name, Plugin pi);
    }
}
```

The `initializePlugins` operation is used in special cases when an application needs to manually initialize one or more plug-ins, as discussed in the next section.

The `getPlugin` operation returns a reference to a specific plug-in. The `name` argument must match an installed plug-in, otherwise the operation raises `NotRegisteredException`. This operation is useful when a plug-in exports an interface that an application can use to query or customize its attributes or behavior.

Finally, `addPlugin` provides a way for an application to install a plug-in directly, without the use of a configuration property. This plug-in's `initialize` operation will be invoked if `initializePlugins` has not yet been called on the plug-in manager. If `initializePlugins` has already been called before a plug-in is added, Ice does not invoke `initialize` on the plug-in, but does invoke `destroy` during communicator destruction.

Delayed Plug-in Initialization

It is sometimes necessary for an application to manually configure a plug-in prior to its initialization. For example, SSL keys are often protected by a passphrase, but a developer may be understandably reluctant to specify that passphrase in a configuration file because it would be exposed in clear text. The developer would likely prefer to configure the IceSSL plug-in with a password callback instead; however, this must be done before the plug-in is initialized and attempts to load the SSL key. The solution is to configure the Ice run time so that it postpones the initialization of its plug-ins:

```
Ice.InitPlugins=0
```

When `Ice.InitPlugins` is set to zero, initializing plug-ins becomes the application's responsibility. The example below demonstrates how to perform this initialization:

C++11 C++98

```
communicator = ...
auto pm = communicator->getPluginManager();
auto ssl = std::dynamic_pointer_cast<IceSSL::Plugin>(pm->getPlugin("Ice
SSL"));
ssl->setPasswordPrompt(...);
pm->initializePlugins();
```

```
communicator = ...
Ice::PluginManagerPtr pm = communicator->getPluginManager();
IceSSL::PluginPtr ssl = IceSSL::PluginPtr::dynamicCast(pm->getPlugin("I
ceSSL"));
ssl->setPasswordPrompt(...);
pm->initializePlugins();
```

After obtaining the IceSSL plug-in and establishing the password callback, the application invokes `initializePlugins` on the plug-in manager object to commence plug-in initialization.

See Also

- IceSSL
- Ice.InitPlugins
- Ice.Plugin.*
- Ice.PluginLoadOrder

Client-Side Features

This section presents all APIs and features that are specific to client applications.

Server-specific features are presented in Server-Side Features, while transverse features best understood while considering both sides of a client-server interaction are described in Client-Server Features.

Topics

Proxies

The introduction to proxies provided in Terminology describes a proxy as a local artifact that makes a remote invocation as easy to use as a regular function call. In fact, processing remote invocations is just one of a proxy's many responsibilities. A proxy also encapsulates the information necessary to contact the object, including its identity and addressing details such as endpoints. Proxy methods provide access to configuration and connection information, and act as factories for creating new proxies. Finally, a proxy initiates the establishment of a new connection when necessary.

Topics

See Also

- Terminology
- Object Identity
- Proxy Methods
- Connection Establishment

Obtaining Proxies

This page describes the ways an application can obtain a proxy.

On this page:

Obtaining a Proxy from a String

Slice

```
module Ice
{
    local interface Communicator
    {
        Object* stringToProxy(string str);
        // ...
    }
}
```

The communicator operation `stringToProxy` creates a proxy from its stringified representation, as shown in the following C++ example:

C++11 C++98

```
auto p = communicator->stringToProxy("ident:tcp -p 5000"); // p is a
std::shared_ptr<Ice::ObjectPrx>
```

```
Ice::ObjectPrx p = communicator->stringToProxy("ident:tcp -p 5000");
```

Obtaining a Proxy from Properties

Slice

```
module Ice
{
    local interface Communicator
    {
        Object* propertyToProxy(string property);
        // ...
    }
}
```

Rather than hard-coding a stringified proxy as the previous example demonstrated, an application can gain more flexibility by externalizing the proxy in a configuration property. For example, we can define a property that contains our stringified proxy as follows:

```
MyApp.Proxy=ident:tcp -p 5000
```

We can use the communicator operation `propertyToProxy` to convert the property's value into a proxy. A null proxy is returned if no property is found with the specified name. For example in Java:

Java Java Compat

```
com.zeroc.Ice.ObjectPrx p = communicator.propertyToProxy("MyApp.Proxy");
```

```
Ice.ObjectPrx p = communicator.propertyToProxy("MyApp.Proxy");
```

As an added convenience, `propertyToProxy` allows you to define subordinate properties that configure the proxy's local settings. The properties below demonstrate this feature:

```
MyApp.Proxy=ident:tcp -p 5000
MyApp.Proxy.PreferSecure=1
MyApp.Proxy.EndpointSelection=Ordered
```

These additional properties simplify the task of customizing a proxy (as you can with proxy methods) without the need to change the application's code. The properties shown above are equivalent to the following statements:

Java Java Compat

```
com.zeroc.Ice.ObjectPrx p = communicator.stringToProxy("ident:tcp -p
5000");
p = p.ice_preferSecure(true);
p =
p.ice_endpointSelection(com.zeroc.Ice.EndpointSelectionType.Ordered);
```

```
Ice.ObjectPrx p = communicator.stringToProxy("ident:tcp -p 5000");
p = p.ice_preferSecure(true);
p = p.ice_endpointSelection(Ice.EndpointSelectionType.Ordered);
```

The list of supported proxy properties includes the most commonly-used proxy settings. The communicator prints a warning by default if it does not recognize a subordinate property. You can disable this warning using the property `Ice.Warn.UnknownProperties`.

Note that proxy properties can themselves have proxy properties. For example, the following sets the `PreferSecure` property on the default locator's router:

```
Ice.Default.Locator.Router.PreferSecure=1
```

Obtaining a Proxy using Factory Methods

Proxy factory methods allow you to modify aspects of an existing proxy. Since proxies are immutable, factory methods always return a new proxy if the desired modification differs from the proxy's current configuration. Consider the following C# example:

C#

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
p = p.ice_oneway();
```

`ice_oneway` is considered a factory method because it returns a proxy configured to use oneway invocations. If the original proxy uses a different invocation mode, the return value of `ice_oneway` is a new proxy object.

The `checkedCast` and `uncheckedCast` methods can also be considered factory methods because they return new proxies that are narrowed to a particular Slice interface. A call to `checkedCast` or `uncheckedCast` typically follows the use of other factory methods, as shown below:

C#

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
Ice.LocatorPrx loc =
Ice.LocatorPrxHelper.checkedCast(p.ice_secure(true));
```

Note however that, once a proxy has been narrowed to a Slice interface, it is not normally necessary to perform another down-cast after using a factory method. For example, we can rewrite this example as follows:

C#

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
Ice.LocatorPrx loc = Ice.LocatorPrxHelper.checkedCast(p);
loc = (Ice.LocatorPrx)p.ice_secure(true);
```

A language-specific cast may be necessary, as shown here for C#, because the factory methods are declared to return the type `ObjectPrx`, but the proxy object itself retains its narrowed type. The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Obtaining a Proxy by Invoking Operations

An application can also obtain a proxy as the result of an Ice invocation. Consider the following Slice definitions:

Slice

```
interface Account { ... }
interface Bank
{
    Account* findAccount(string id);
}
```

Invoking the `findAccount` operation returns a proxy for an `Account` object. There is no need to use `checkedCast` or `uncheckedCast` on this proxy because it has already been narrowed to the `Account` interface. The C++ code below demonstrates how to invoke `findAccount`:

C++11 C++98

```
std::shared_ptr<BankPrx> bank = ...
auto acct = bank->findAccount(id); // acct is a shared_ptr<AccountPrx>
```

```
BankPrx bank = ...
AccountPrx acct = bank->findAccount(id);
```

Of course, the application must have already obtained a proxy for the bank object using one of the techniques shown above.

See Also

- Communicator
- Proxy and Endpoint Syntax
- Proxy Methods
- Proxy Properties
- Ice.Warn.*

Converting Proxies to Strings

This page describes how an application can *stringify* a proxy and create a property dictionary that captures all the proxy's properties.

On this page:

Stringifying a Proxy

Slice

```
module Ice
{
    local interface Communicator
    {
        string proxyToString(Object* obj);
        // ...
    }
}
```

The communicator operation `proxyToString` converts a proxy into its stringified representation. Instead of calling `proxyToString` on the communicator, you can use the `ice_toString` proxy method to stringify it. However, you can only stringify non-null proxies that way — to stringify a null proxy, you must use `proxyToString`. (The stringified representation of a null proxy is the empty string.)

`proxyToString` and `ice_toString` stringify non-printable ASCII characters and non-ASCII characters in the proxy's identity, facet and object adapter id as specified through the `Ice.ToStringMode` property.

For example:

C++

```
prx = ...
string str = commmunicator->proxyToString(prx);

// is equivalent to:
string str2;
if(prx)
{
    str2 = prx->ice_toString();
}
```

proxyToProperty

Slice

```
module Ice
{
    dictionary<string, string> PropertyDict;
    local interface Communicator
    {
        PropertyDict proxyToProperty(Object* proxy, string property);
        // ...
    }
}
```

The `proxyToProperty` operation returns the set of proxy properties for the supplied proxy. The `property` parameter specifies the base name for the properties in the returned set.

See Also

- Obtaining Proxies
- Proxy and Endpoint Syntax
- Proxy Methods
- Proxy Properties

Proxy Methods

Although the core proxy functionality is supplied by a language-specific base class, we can describe the proxy methods in terms of Slice operations as shown below:

Slice

```
bool ice_isA(string id);
void ice_ping();
StringSeq ice_ids();
string ice_id();
Communicator ice_getCommunicator();
string ice_toString();
Object* ice_identity(Identity id);
Identity ice_getIdentity();
Object* ice_adapterId(string id);
string ice_getAdapterId();
Object* ice_endpoints(EndpointSeq endpoints);
EndpointSeq ice_getEndpoints();
Object* ice_endpointSelection(EndpointSelectionType t);
EndpointSelectionType ice_getEndpointSelection();
Object* ice_context(Context ctx);
Context ice_getContext();
Object* ice_facet(string facet);
string ice_getFacet();
Object* ice_twoway();
bool ice_isTwoway();
Object* ice_oneway();
bool ice_isOneway();
Object* ice_batchOneway();
bool ice_isBatchOneway();
Object* ice_datagram();
bool ice_isDatagram();
Object* ice_batchDatagram();
bool ice_isBatchDatagram();
Object* ice_secure(bool b);
bool ice_isSecure();
EncodingVersion ice_getEncodingVersion();
Object* ice_encodingVersion(EncodingVersion v);
Object* ice_preferSecure(bool b);
bool ice_isPreferSecure();
Object* ice_compress(bool b);
optional(1) bool ice_getCompress();
Object* ice_timeout(int timeout);
optional(1) int ice_getTimeout();
Object* ice_router(Router* rtr);
Router* ice_getRouter();
Object* ice_locator(Locator* loc);
Locator* ice_getLocator();
Object* ice_locatorCacheTimeout(int seconds);
int ice_getLocatorCacheTimeout();
```

```
Object* ice_collocationOptimized(bool b);
bool ice_isCollocationOptimized();
Object* ice_invocationTimeout(int timeout);
int ice_getInvocationTimeout();
Object* ice_connectionId(string id);
string ice_getConnectionId();
Object* ice_fixed(Connection con);
Connection ice_getConnection();
Connection ice_getCachedConnection();
Object* ice_connectionCached(bool b);
bool ice_isConnectionCached();
void ice_flushBatchRequests();
bool ice_invoke(string operation, OperationMode mode,
                ByteSeq inParams, out ByteSeq outParams);

// Only with C# mapping
```

```
System.Threading.Tasks.TaskScheduler ice_scheduler();
// Only with Java mapping
java.util.concurrent.Executor ice_executor();
```

These methods can be categorized as follows:

- Remote inspection: methods that return information about the remote object. These methods make remote invocations and therefore accept an optional trailing argument of type `Ice::Context`.
- Local inspection: methods that return information about the proxy's local configuration.
- Factory: methods that return new proxy instances configured with different features.
- Request processing: methods that flush batch requests and send "dynamic" Ice invocations.

Proxies are immutable, so factory methods allow an application to obtain a new proxy with the desired configuration. Factory methods essentially clone the original proxy and modify one or more features of the new proxy.

Many of the factory methods are not supported by fixed proxies, which are used in conjunction with bidirectional connections. Attempting to invoke one of these methods causes the Ice run time to raise `FixedProxyException`.

The core proxy methods are explained in greater detail in the following table:

Method	Description	Remote
`ice_isA`	Returns `true` if the remote object supports the type indicated by the `id` argument, otherwise `false`. This method can only be invoked on a twoway proxy.	Yes
`ice_ping`	Determines whether the remote object is reachable. Does not return a value.	Yes
`ice_ids`	Returns the type IDs of the types supported by the remote object. The return value is an array of strings. This method can only be invoked on a twoway proxy.	Yes
`ice_id`	Returns the type ID of the most-derived type supported by the remote object. This method can only be invoked on a twoway proxy.	Yes
`ice_getCommunicator`	Returns the communicator that was used to create this proxy.	No
`ice_toString`	Returns the string representation of the proxy. The `Ice.ToStringMode` property controls how non-printable ASCII characters and non-ASCII characters in the identity, facet and object adapter id of this proxy are represented in the resulting string. When called on a fixed proxy, this method returns a stringified proxy without endpoints.	No
`ice_identity`	Returns a new proxy having the given identity.	No
`ice_getIdentity`	Returns the identity of the Ice object represented by the proxy.	No
`ice_adapterId`	Returns a new proxy having the given adapter ID.	No
`ice_getAdapterId`	Returns the proxy's adapter ID, or an empty string if no adapter ID is configured (as is the case with direct proxies).	No
`ice_endpoints`	Returns a new proxy having the given endpoints.	No
`ice_getEndpoints`	Returns a sequence of `Endpoint` objects representing the direct proxy's endpoints. For indirect proxies, an empty sequence is returned.	No
`ice_endpointSelection`	Returns a new proxy having the given endpoint selection policy (random or ordered).	No
`ice_getEndpointSelection`	Returns the endpoint selection policy for the proxy.	No
`ice_context`	Returns a new proxy having the given request context.	No
`ice_getContext`	Returns the request context associated with the proxy.	No
`ice_facet`	Returns a new proxy having the given facet name.	No
`ice_getFacet`	Returns the name of the facet associated with the proxy, or an empty string if no facet has been set.	No

`ice_twoway`	Returns a new proxy for making twoway invocations.	No
`ice_isTwoway`	Returns `true` if the proxy uses twoway invocations, otherwise `false`.	No
`ice_oneway`	Returns a new proxy for making oneway invocations.	No
`ice_isOneway`	Returns `true` if the proxy uses oneway invocations, otherwise `false`.	No
`ice_batchOneway`	Returns a new proxy for making batch oneway invocations.	No
`ice_isBatchOneway`	Returns `true` if the proxy uses batch oneway invocations, otherwise `false`.	No
`ice_datagram`	Returns a new proxy for making datagram invocations.	No
`ice_isDatagram`	Returns `true` if the proxy uses datagram invocations, otherwise `false`.	No
`ice_batchDatagram`	Returns a new proxy for making batch datagram invocations.	No
`ice_isBatchDatagram`	Returns `true` if the proxy uses batch datagram invocations, otherwise `false`.	No
`ice_secure`	Returns a new proxy whose endpoints may be filtered depending on the boolean argument. If `true`, only endpoints using secure transports are allowed, otherwise all endpoints are allowed.	No
`ice_getEncodingVersion`	Returns the encoding version that is used to encode requests invoked on this proxy.	No
`ice_encodingVersion`	Returns a new proxy that uses the given encoding version to encode requests. Raises `UnsupportedEncodingException` if the version is invalid.	No
`ice_isSecure`	Returns `true` if the proxy uses only secure endpoints, otherwise `false`.	No
`ice_preferSecure`	Returns a new proxy whose endpoints are filtered depending on the boolean argument. If `true`, endpoints using secure transports are given precedence over endpoints using non-secure transports. If `false`, the default behavior gives precedence to endpoints using non-secure transports.	No
`ice_isPreferSecure`	Returns `true` if the proxy prefers secure endpoints, otherwise `false`.	No
`ice_compress`	Returns a new proxy whose protocol compression capability is determined by the boolean argument. If `true`, the proxy uses protocol compression if it is supported by the endpoint. If `false`, protocol compression is never used. This setting overrides the compression setting from the proxy endpoints.	No
`ice_getCompress`	Returns whether or not a compression override is set on the proxy. If no optional value is present, no override is set. Otherwise, the value is true if compression is enabled, false otherwise.	
`ice_timeout`	Returns a new proxy whose endpoints all have the given connection timeout value in milliseconds. A value of `-1` disables timeouts. This setting overrides the timeout from the proxy endpoints.	No
`ice_getTimeout`	Returns whether or not a timeout override is set on the proxy. If no optional value is present, no override is set. Otherwise, the optional value contains the timeout override.	
`ice_router`	Returns a new proxy configured with the given router proxy.	No
`ice_getRouter`	Returns the router that is configured for the proxy (null if no router is configured).	No
`ice_locator`	Returns a new proxy with the specified locator.	No
`ice_getLocator`	Returns the locator that is configured for the proxy (null if no locator is configured).	No
`ice_locatorCacheTimeout`	Returns a new proxy with the specified locator cache timeout in seconds. When binding a proxy to an endpoint, the run time caches the proxy returned by the locator and uses the cached proxy while the cached proxy has been in the cache for less than the timeout. Proxies older than the timeout cause the run time to rebind via the locator. A value of 0 disables caching entirely, and a value of -1 means that cached proxies never expire. The default value is -1.	No

`ice_getLocatorCacheTimeout`	Returns the locator cache timeout value in seconds.	No
`ice_collocationOptimized`	Returns a new proxy configured for collocation optimization. If `true`, collocated optimizations are enabled. The default value is `true`.	No
`ice_isCollocationOptimized`	Returns `true` if the proxy uses collocation optimization, otherwise `false`.	No
`ice_invocationTimeout`	Returns a new proxy configured with the specified invocation timeout in milliseconds. A value of `-1` means an invocation never times out. A value of `-2` provides backward compatibility with Ice versions prior to 3.6 by disabling invocation timeouts and using connection timeouts to wait for the response of an invocation.	No
`ice_getInvocationTimeout`	Returns the invocation timeout value in milliseconds.	No
`ice_connectionId`	Returns a new proxy having the given connection ID.	No
`ice_getConnectionId`	Returns the connection ID, or an empty string if no connection ID has been configured.	No
`ice_fixed`	Returns a new fixed proxy bound to the given connection.	
`ice_getConnection` `ice_getConnectionAsync`	Returns an object representing the connection used by the proxy. If the proxy is not currently associated with a connection, the Ice run time attempts to establish a connection first, which means this method can potentially block while network operations are in progress. Use the asynchronous method to avoid blocking.	No
`ice_getCachedConnection`	Returns an object representing the connection used by the proxy, or null if the proxy is not currently associated with a connection.	No
`ice_connectionCached`	Enables or disables connection caching for the proxy.	No
`ice_isConnectionCached`	Returns `true` if the proxy uses connection caching, otherwise `false`.	No
`ice_flushBatchRequests`	Sends a batch of operation invocations synchronously or asynchronously.	Yes
`ice_invoke`	Allows dynamic invocation of an operation without the need for compiled Slice definitions. Requests can be sent synchronously or asynchronously.	Yes
`ice_scheduler`	This method is only available in the C# mapping. It returns a System.Threading.Tasks.TaskScheduler object that uses the Ice thread pool to execute tasks.	No
`ice_executor`	This method is only available in the Java mapping. It returns a java.util.concurrent.Executor object that uses the Ice thread pool to execute tasks.	No

See Also

- Request Contexts
- Oneway Invocations
- Batched Invocations
- Versioning
- Connection Timeouts
- Connection Establishment
- Using Connections
- Dynamic Invocation and Dispatch
- Bidirectional Connections
- Glacier2
- Data Encoding

Proxy Endpoints

Proxy endpoints are the client-side equivalent of object adapter endpoints. A proxy endpoint identifies the protocol information used to contact a remote object, as shown in the following example:

```
tcp -h frosty.zeroc.com -p 10000
```

This endpoint states that an object is reachable via TCP on the host `frosty.zeroc.com` and the port `10000`.

A proxy must have, or be able to obtain, at least one endpoint in order to be useful. A *direct proxy* contains one or more endpoints:

```
MyObject:tcp -h frosty.zeroc.com -p 10000:ssl -h frosty.zeroc.com -p
10001
```

In this example the object with the identity `MyObject` is available at two separate endpoints, one using TCP and the other using SSL.

If a direct proxy does not contain the `-h` option (that is, no host is specified), the Ice run time uses the value of the `Ice.Default.Host` property. If `Ice.Default.Host` is not defined, the localhost interface is used.

An *indirect proxy* uses a locator to retrieve the endpoints dynamically. One style of indirect proxy contains an adapter identifier:

```
MyObject @ MyAdapter
```

When this proxy requires the endpoints associated with `MyAdapter`, it requests them from the locator.

The other style of indirect proxy is a proxy with just an object identity, called a well-known proxy:

```
MyObject
```

See Also

- Terminology
- Object Adapter Endpoints
- Locators

Filtering Proxy Endpoints

A proxy's configuration determines how its endpoints are used. For example, a proxy configured for secure communication will only use endpoints having a secure protocol, such as SSL.

The factory methods listed in the table below allow applications to manipulate endpoints indirectly. Calling one of these methods returns a new proxy whose endpoints are used in accordance with the proxy's new configuration.

Method	Description
`ice_secure`	Selects only endpoints using a secure protocol (e.g., SSL).
`ice_datagram`	Selects only endpoints using a datagram protocol (e.g., UDP).
`ice_batchDatagram`	Selects only endpoints using a datagram protocol (e.g., UDP).
`ice_twoway`	Selects only endpoints capable of making twoway invocations (e.g., TCP, SSL). For example, this disables datagram endpoints.
`ice_oneway`	Selects only endpoints capable of making reliable oneway invocations (e.g., TCP, SSL). For example, this disables datagram endpoints.
`ice_batchOneway`	Selects only endpoints capable of making reliable oneway batch invocations (e.g., TCP, SSL). For example, this disables datagram endpoints.

Upon return, the set of endpoints in the new proxy is unchanged from the old one. However, the new proxy's configuration drives a filtering process that the Ice run time performs during connection establishment.

The factory methods do not raise an exception if they produce a proxy with no viable endpoints. For example, the C++ statement below creates such a proxy:

C++

```
proxy = communicator->stringToProxy("id:tcp -p 10000")->ice_datagram();
```

It is always possible that a proxy could become viable after additional factory methods are invoked, therefore the Ice run time does not raise an exception until connection establishment is attempted. At that point, the application can expect to receive `NoEndpointException` if the filtering process eliminates all endpoints.

An application can also create a proxy with a specific set of endpoints using the `ice_endpoints` factory method, whose only argument is a sequence of `Ice::Endpoint` objects. At present, an application is not able to create new instances of `Ice::Endpoint`, but rather can only incorporate instances obtained by calling `ice_getEndpoints` on a proxy. Note that `ice_getEndpoints` may return an empty sequence if the proxy has no endpoints, as is the case with an indirect proxy.

See Also

- Proxy Endpoints
- Proxy Methods
- Connection Establishment

Proxy Defaults and Overrides

It is important to understand how proxies are influenced by Ice configuration properties and settings. The relevant properties can be classified into two categories: defaults and overrides.

On this page:

Proxy Default Properties

Default properties affect proxies created as the result of an Ice invocation, or by calling `stringToProxy` or `propertyToProxy` on a communicator. These properties do not influence proxies created by proxy factory methods.

For example, suppose we define the following default property:

```
Ice.Default.PreferSecure=1
```

We can verify that the property has the desired affect using the following C++ code:

C++11 C++98

```
auto p = communicator->stringToProxy(...);
assert(p->ice_isPreferSecure());
```

```
Ice::ObjectPrx p = communicator->stringToProxy(...);
assert(p->ice_isPreferSecure());
```

Furthermore, we can verify that the property does not affect proxies returned by factory methods:

C++11

```
auto p2 = p->ice_preferSecure(false);
assert(!p2->ice_isPreferSecure());
auto p3 = p2->ice_oneway();
assert(!p3->ice_isPreferSecure());
```

Proxy Override Properties

Defining an override property causes the Ice run time to ignore any equivalent proxy setting and use the override property value instead. For example, consider the following property definition:

```
Ice.Override.Secure=1
```

This property instructs the Ice run time to use only secure endpoints, producing the same semantics as calling `ice_secure(true)` on every proxy. However, the property does not alter the settings of an existing proxy, but rather directs the Ice run time to use secure endpoints regardless of the proxy's security setting. We can verify that this is the case using the following C++ code:

C++11 C++98

```
auto p = communicator->stringToProxy(...); // p is a
shared_ptr<Ice::ObjectPrx>
p = p->ice_secure(false);
assert(!p->ice_isSecure()); // The security setting is retained.
```

```
Ice::ObjectPrx p = communicator->stringToProxy(...);
p = p->ice_secure(false);
assert(!p->ice_isSecure()); // The security setting is retained.
```

See Also

- Communicator
- Ice.Default.*
- Ice.Override.*

Proxy-Based Load Balancing

Ice supports two types of load balancing:

- Locator-based load balancing
- Proxy-based load balancing

An application can use only one type or combine both to achieve the desired behavior. This page discusses proxy-based load balancing; refer to Locator Semantics for Clients for information on load balancing with a locator.

On this page:

Proxy Connection Caching

Before we discuss load balancing, it's important to understand the relationship between proxies and connections.

By default a proxy remembers its connection and uses it for all invocations until that connection is closed. You can prevent a proxy from caching its connection by calling the `ice_connectionCached` proxy method with an argument of false. Once connection caching is disabled, each invocation on a proxy causes the Ice run time to execute its connection establishment process.

Note that each invocation on such a proxy does not necessarily cause the Ice run time to establish a new connection. It only means that the Ice run time does not assume that it can reuse the connection of the proxy's previous invocation. Whether the Ice run time actually needs to establish a new connection for the next invocation depends on several factors.

As with any feature, you should only use it when the benefits outweigh the risks. With respect to a proxy's connection caching behavior, there is certainly a small amount of computational overhead associated with executing the connection establishment process for each invocation, as well as the risk of significant overhead each time a new connection is actually created.

Proxies with Multiple Endpoints

A proxy can contain zero or more endpoints. A proxy with no endpoints typically means it's an indirect proxy that requires a location service t o convert the proxy's symbolic information into endpoints at run time. Otherwise, a direct proxy contains at least one endpoint.

Regardless of whether endpoints are specified directly or indirectly, a proxy having multiple endpoints implies at a minimum that the target object is available via multiple network interfaces, but often also means the object is replicated to improve scalability and reliability. Such a proxy provides a client with several load balancing options even when no location service is involved. The proxy's own configuration drives the run-time behavior, depending on how the client configures the proxy's endpoint selection type and connection caching settings.

For example, suppose that a proxy contains several endpoints. In its default configuration, a proxy uses the `Random` endpoint selection type and caches its connection. Upon the first invocation, the Ice run time selects one of the proxy's endpoints at random and uses that connection for all subsequent invocations until the connection is closed. For some applications, this form of load balancing may be sufficient.

Suppose now that we use the `Ordered` endpoint selection type instead. In this case, the Ice run time always attempts to establish connections using the endpoints in the order they appear in the proxy. Normally an application uses this configuration when there is a preferred order to the servers. Again, once connected, the Ice run time uses whichever connection was chosen indefinitely.

Per-Request Load Balancing

When we disable the connection caching behavior of a proxy with multiple endpoints, its semantics undergo a significant change. Using the `Random` endpoint selection type, the Ice run time selects one of the endpoints at random and establishes a connection to it if one is not already established, and this process is repeated *prior to each subsequent invocation*. This is called *per-request load balancing* because each request can potentially be directed to a different server.

> Using the `Ordered` endpoint selection type is not as common in this scenario; its main purpose would be to fall back on a secondary server if the primary server is not available, but it causes the Ice run time to attempt to contact the primary server during each request.

Here's some code that shows how to configure the proxy:

C++11 C++98

```
auto proxy = communicator->stringToProxy("hello:tcp -h 10.0.0.1 -p
2000:tcp -h 10.0.0.2 -p 2001");
proxy = proxy->ice_connectionCached(false);
proxy = proxy->ice_endpointSelection(Ice::Random);
// If also using a locator:
proxy = proxy->ice_locatorCacheTimeout(...);
```

```
Ice::ObjectPrx proxy = communicator->stringToProxy("hello:tcp -h
10.0.0.1 -p 2000:tcp -h 10.0.0.2 -p 2001");
proxy = proxy->ice_connectionCached(false);
proxy = proxy->ice_endpointSelection(Ice::Random);
// If also using a locator:
proxy = proxy->ice_locatorCacheTimeout(...);
```

We create a proxy with two endpoints, then use the factory methods to disable connection caching and set the endpoint selection type. The `Random` setting means that, after we've made a few invocations with the proxy, the Ice run time will have established connections to both endpoints. After incurring the initial expense of opening the connections, each subsequent invocation will randomly use one of the existing connections.

If you're also using a location service, you may want to modify the proxy's locator cache timeout to force the Ice run time in the client to query the locator more frequently.

See Also

- Terminology
- Proxy Methods
- Connection Establishment
- Active Connection Management
- Ice.Default.*
- Locator Semantics for Clients
- Load Balancing

Indirect Proxy with Object Adapter Identifier

A proxy with an object adapter identifier (*@adapterId* in stringified form) is a form of indirect proxy. Such a proxy consists of an object identity, an object adapter identifier and (optionally) proxy options such as -t (for two-way proxies), as described on Proxy and Endpoint Syntax.

For example:

```
Root@fsadapter      # proxy to Root object hosted by object adapter
fsadapter (two-way proxy by default)
Root@fsadapter -o   # oneway proxy
```

When you invoke an operation on such an indirect proxy, Ice first *resolves* the object adapter identifier–Ice checks if it corresponds to a local object adapter, or retrieves the endpoints published by this object adapter.

The resolution proceeds as follows:

1. If collocation optimization is enabled (the default), Ice checks if an object adapter associated with the same communicator as the proxy has the desired object adapter identifier (set through `ReplicaGroupId` or `AdapterId`). If there is such an object adapter, Ice then sends requests to this object adapter using collocation optimization. The holding state of the object adapters is ignored for this search and subsequent collocated dispatches.
2. Otherwise, if no local object adapters carries the desired object adapter identifier (or collocation optimization is disabled), and a locator is configured with the communicator:
    a. Ice looks up this object adapter identifier in its locator cache.
    b. If this lookup fails, Ice resolves this object adapter identifier using the locator.
3. In case the preceding steps can't locate the object adapter, the invocation fails with `NoEndpointException`.

See Also

- Locators
- Well-Known Proxies

Well-Known Proxy

A proxy with no endpoint or object adapter identifier (*@adapterId* in stringified form) is called a well-known proxy. A well-known proxy consists of an object identity plus (optionally) proxy options such as -t (for two-way proxies), as described on Proxy and Endpoint Syntax. Well-known proxies are a form of indirect proxies.

For example:

```
Root       # well-known proxy to Root object (two-way by default)
Root -o    # oneway proxy
```

When you invoke an operation on a well-known proxy, Ice locates the target object as follows:

1. If collocation optimization is enabled (the default), Ice looks up the object identity in the Active Servant Map (ASM) of all object adapters associated with the same communicator as this well-known proxy. The servant locators and default servants registered with these object adapters are not consulted. If the object is found in one of these ASMs, Ice then sends requests to this object using collocation optimization. The holding state of the object adapter is ignored for this search and subsequent collocated dispatches to the servant.
2. Otherwise, if Ice does not find this object identity in one of these local ASMs (or collocation optimization is disabled), and a locator is configured with the communicator:
    a. Ice looks up this object identity in its locator cache.
    b. If this lookup fails, Ice resolves this object identity using the locator.
3. In case the preceding steps can't locate the target object or endpoints, the invocation fails with `NoEndpointException`.

See Also

- Locators
- Well-Known Objects

Proxy and Endpoint Syntax

On this page:

Syntax for Stringified Proxies

Synopsis

identity -f *facet* -e *encoding* -p *protocol* -t -o -O -d -D -s @ *adapter_id* : *endpoints*

Description

A stringified proxy consists of an identity, proxy options, and an optional object adapter identifier or endpoint list. White space (the space, tab (\t), line feed (\n), and carriage return (\r) characters) act as token delimiters; if a white space character appears as part of a component of a stringified proxy (such as the identity), it must be quoted or escaped as described below.

A proxy containing an identity with no endpoints is a well-known proxy; a proxy with an identity and an object adapter identifier represents an indirect proxy that will be resolved using the Ice locator.

Proxy options configure the invocation mode:

-f *facet*	Select a facet of the Ice object.
-e *encoding*	Specify the Ice encoding version to use when an invocation on this proxy marshals parameters.
-p *protocol*	Specify the Ice protocol version to use when sending a request with this proxy.
-t	Configures the proxy for twoway invocations (default).
-o	Configures the proxy for oneway invocations.
-O	Configures the proxy for batch oneway invocations.
-d	Configures the proxy for datagram invocations.
-D	Configures the proxy for batch datagram invocations.
-s	Configures the proxy for secure invocations.

The proxy options -t, -o, -O, -d, and -D are mutually exclusive.

The object identity identity is structured as [*category*/]*name*, where the *category* component and slash separator are optional. If *identity* contains white space or either of the characters : or @, it must be enclosed in single or double quotes. The *category* and *name* components are strings that are encoded as described in Object Identity, in particular, any occurrence of a slash (/) in *category* or *name* must be escaped with a backslash (i.e., \/).

The *facet* argument of the -f option represents a facet name. If *facet* contains white space, it must be enclosed in single or double quotes. A facet name is a string encoded like a Slice String Literal.

Likewise, an object adapter identifier *adapter_id* is a string encoded like a Slice String Literal. If *adapter_id* contains white space, it must be enclosed in single or double quotes.

Single or double quotes can be used to prevent white space characters from being interpreted as delimiters. Double quotes prevent interpretation of a single quote as an opening or closing quote, for example:

```
"a string with a ' quote"
```

Single quotes prevent interpretation of a double quote as an opening or closing quote. For example:

```
'a string with a " quote'
```

If *endpoints* are specified, they must be separated with a colon (`:`) and formatted as described in the endpoint syntax. The order of endpoints in the stringified proxy is not necessarily the order in which connections are attempted during binding: when a stringified proxy is converted into a proxy instance, by default, the endpoint list is randomized as a form of load balancing. You can change this default behavior using the properties `Ice.Default.EndpointSelection` and *name*`.EndpointSelection`.

If the `-s` option is specified, only those endpoints that support secure invocations are considered during binding. If no valid endpoints are found, the application receives `Ice::NoEndpointException`.

Otherwise, if the `-s` option is not specified, the endpoint list is ordered so that non-secure endpoints have priority over secure endpoints during binding. In other words, connections are attempted on all non-secure endpoints before any secure endpoints are attempted.

The `-e` and `-p` options specify the encoding and protocol versions supported by the target object, respectively. The Ice run time in the client is responsible for using a compatible version of the encoding or protocol, where *compatible* means the same major version and a minor version that is equal to or less than the specified minor version. If the `-e` option is not defined, the proxy uses the default version specified by `Ice.Default.EncodingVersion`.

If an unknown option is specified, or the stringified proxy is malformed, the application receives `Ice::ProxyParseException`. If an endpoint is malformed, the application receives `Ice::EndpointParseException`.

Syntax for Stringified Endpoints

Synopsis

endpoint `:` *endpoint*

Description

An endpoint list comprises one or more endpoints separated by a colon (`:`).

An endpoint has the following format:

protocol option

The supported protocols are `tcp`, `udp`, `ssl`, `ws`, `wss`, and `default`. If `default` is used, it is replaced by the value of the `Ice.Default.Protocol` property. If an endpoint is malformed, or an unknown protocol is specified, the application receives `Ice::EndpointParseException`. The `ssl` and `wss` protocols are only available if the IceSSL plug-in is installed.

Ice uses endpoints for two similar but distinct purposes:

1. In a client context (that is, in a proxy), endpoints determine how Ice establishes a connection to a server.
2. In a server context (that is, in an object adapter's configuration), endpoints define the addresses and transports over which new incoming connections are accepted. These endpoints are also embedded in the proxies created by the object adapter, unless a separate set of "published" endpoints are explicitly configured.

The sections that follow discuss the addressing component of endpoints, as well as the protocols and their supported options.

> See Object Adapter Endpoints for examples.

Address Syntax

Synopsis

```
host : hostname | x.x.x.x (IPv4)
host : hostname | ":x:x:x:x:x:x:x" (IPv6)
```

Description

Ice supports Internet Protocol (IP) versions 4 and 6 in all language mappings.

Support for these protocols is configured using the properties `Ice.IPv4` and `Ice.IPv6` (both enabled by default).

In the endpoint descriptions below, the *host* parameter represents either a host name that is resolved via the Domain Name System (DNS), an IPv4 address in dotted quad notation, or an IPv6 address in 128-bit hexadecimal format and enclosed in double quotes. Due to limitation of the DNS infrastructure, host and domain names are restricted to the ASCII character set.

The presence (or absence) of the *host* parameter has a significant influence on the behavior of the Ice run time. The table below describes these semantics:

Value	Client Semantics	Server Semantics
None	If *host* is not specified in a proxy, Ice uses the value of the `Ice.Default.Host` property. If that property is not defined, outgoing connections are only attempted over loopback interfaces.	If *host* is not specified in an object adapter endpoint, Ice uses the value of the `Ice.Default.Host` property. If that property is not defined, the adapter behaves as if the wildcard symbol * was specified (see below).
Host name	The host name is resolved via DNS. Outgoing connections are attempted to each address returned by the DNS query.	The host name is resolved via DNS, and the object adapter listens on the network interfaces corresponding to each address returned by the DNS query. The specified host name is embedded in proxies created by the adapter if the endpoint specifies a fixed port. Otherwise, if the port is selected by the operating system, proxies will embed an endpoint for each address and the system allocated port.
IPv4 address	An outgoing connection is attempted to the given address.	The object adapter listens on the network interface corresponding to the address. The specified address is embedded in proxies created by the adapter.
IPv6 address	An outgoing connection is attempted to the given address.	The object adapter listens on the network interface corresponding to the address. The specified address is embedded in proxies created by the adapter.
`0.0.0.0` (IPv4)	A "wildcard" IPv4 address that causes Ice to try all local interfaces when establishing an outgoing connection.	The adapter listens on all IPv4 network interfaces (including the loopback interface), that is, binds to INADDR_ANY for IPv4. Endpoints for all addresses except loopback are published in proxies (unless loopback is the only available interface, in which case only loopback is published).
`"::"` (IPv6)	A "wildcard" IPv6 address that causes Ice to try all local interfaces when establishing an outgoing connection.	Equivalent to * (see below).
* (IPv4, IPv6)	Not supported in proxies.	The adapter listens on all network interfaces (including the loopback interface), that is, binds to INADDR_ANY for the enabled protocols (IPv4 and/or IPv6). Endpoints for all addresses except loopback and IPv6 link-local are published in proxies (unless loopback is the only available interface, in which case only loopback is published).

There is one additional benefit in specifying a wildcard address for *host* (or not specifying it at all) in an object adapter's endpoint: if the list of network interfaces on a host may change while the application is running, using a wildcard address for *host* ensures that the object adapter automatically includes the updated interfaces. Note however that the list of published endpoints is not changed automatically; rather, the application must explicitly refresh the object adapter's endpoints. For diagnostic purposes, you can set the configuration property `Ice.Trace.Network=3` to cause Ice to log the current list of local addresses that it is substituting for the wildcard address.

When IPv4 and IPv6 are enabled, an object adapter endpoint that uses an IPv6 (or * wildcard) address can accept both IPv4 and IPv6 connections.

Java's default network stack always accepts both IPv4 and IPv6 connections regardless of the settings of `Ice.IPv6`. Thus, in Java, an object adapter endpoint that uses the IPv4 wildcard will accept both IPv4 and IPv6 connections. You can configure the Java run time to use only IPv4 by starting your application with the following JVM option:

Java

```
java -Djava.net.preferIPv4Stack=true ...
```

TCP Endpoint Syntax

Synopsis

`tcp -h` *`host`* `-p` *`port`* `-t` *`timeout`* `-z --sourceAddress` *`addr`*

Availability

The TCP transport is a built-in transport, it's always available.

Description

A `tcp` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
`-h` *`host`*	Specifies the host name or IP address of the endpoint. If not specified, the value of `Ice.Default.Host` is used instead.	See Address Syntax.	See Address Syntax.
`-p` *`port`*	Specifies the port number of the endpoint.	Determines the port to which a connection attempt is made (required).	The port will be selected by the operating system if this option is not specified or *`port`* is zero.
`-t` *`timeout`*	Specifies the endpoint timeout in milliseconds.	The value for *`timeout`* must either be `infinite` to specify no timeout, or an integer greater than zero representing the timeout in milliseconds used by the client to open or close connections and to read or write data. If a timeout occurs, the application receives `Ice::TimeoutException`. If this option is not specified, it defaults to the value of `Ice.Default.Timeout`.	The value for *`timeout`* must either be `infinite` to specify no timeout, or an integer greater than zero representing the timeout in milliseconds used by the server to accept or close connections and to read or write data (see Timeouts in Object Adapter Endpoints and Connection Timeouts). *`timeout`* also controls the timeout that is published in proxies created by the object adapter. If this option is not specified, it defaults to the value of `Ice.Default.Timeout`.
`-z`	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.
`--sourceAddress` *`ADDR`*	Specifies the source address used by the connection.	The value for *`ADDR`* must be a numeric IPv4 or IPv6 address. If this option is not specified, it defaults to the value of `Ice.Default.SourceAddress`. This option allows to specify the source address set in the IP packet. It doesn't necessarily imply that the operating system will use the network interface matching this IP address to send out the IP packet. This feature is not supported on Universal Windows (UWP).	Not supported

UDP Endpoint Syntax

Synopsis

```
udp -h host -p port -z --ttl TTL --interface INTF --sourceAddress addr
```

Availability

The UDP transport is a built-in transport except for C++ and Objective-C static builds where it's only available to programs that explicitly register the UDP plug-in. See Using Plugins with Static Libraries.

Description

A `udp` endpoint supports either unicast or multicast delivery; the address resolved by *`host`* argument determines the delivery mode. To use multicast in IPv4, select an IP address in the range `233.0.0.0 to 239.255.255.255`. In IPv6, use an address that begins with `ff`, such as `ff01::1:1`.

A `udp` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
`-h` *`host`*	Specifies the host name or IP address of the endpoint. If not specified, the value of `Ice.Default.Host` is used instead.	See Address Syntax.	See Address Syntax.
`-p` *`port`*	Specifies the port number of the endpoint.	Determines the port to which datagrams are sent (required).	The port will be selected by the operating system if this option is not specified or port is zero.
`-z`	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.
`--ttl` *`TTL`*	Specifies the time-to-live (also known as "hops") of multicast messages.	Determines whether multicast messages are forwarded beyond the local network. If not specified, or the value of *`TTL`* is `-1`, multicast messages are not forwarded. The maximum value is `255`.	N/A
`--interface` *`INTF`*	Specifies the network interface or group for multicast messages (see below).	Selects the network interface for outgoing multicast messages. If not specified, multicast messages are sent using the default interface.	Selects the network interface to use when joining the multicast group. If set to * or not specified, the group is joined on all the local network interfaces.
`--sourceAddress` *`ADDR`*	Binds outgoing socket connections to the network interface associated with *`ADDR`*.	The value for *`ADDR`* must be a numeric IPv4 or IPv6 address. If this option is not specified, it defaults to the value of `Ice.Default.SourceAddress`. This option allows to specify the source address set in the IP packet. It doesn't necessarily imply that the operating system will use the network interface matching this IP address to send out the IP packet. This feature is not supported on Universal Windows (UWP).	Not supported

> **Deprecated options**
> With the 1.0 encoding, UDP endpoints supported 2 additional options: the -e *major.minor* and -v *major.minor* options. These 2 options specified which encoding and protocol was supported by the endpoint. These two options are deprecated with the 1.1 encoding and are ignored (a deprecation warning will be emitted when parsed by the communicator `stringToProxy` method). The supported protocol and encoding is specified on the proxy with the 1.1 encoding.

Multicast Interfaces

When *`host`* denotes a multicast address, the `--interface` *`INTF`* option selects a particular network interface to be used for communication. The format of *`INTF`* depends on the language and IP version:

- C++ and .NET
  `INTF` can be an interface name, such as `eth0`, or an IP address. If using IPv6, it can also be an interface index. Interface names on Windows may contain spaces, such as `Local Area Connection`, therefore they must be enclosed in double quotes.

- Java
  INTF can be an interface name, such as `eth0`, or an IP address. On Windows, Java maps interface names to Unix-style nicknames.

SSL Endpoint Syntax

Synopsis

```
ssl -h host -p port -t timeout -z --sourceAddress addr
```

Availability

The SSL transport is provided by a separate IceSSL plug-in.

Description

An `ssl` endpoint supports the same options as for `tcp` endpoints.

WS Endpoint Syntax

Synopsis

```
ws -r resource -h host -p port -t timeout -z --sourceAddress addr
```

Availability

The WS transport is a built-in transport except for C++ and Objective-C static builds where it's only available to programs that explicitly register the WS plug-in. See Using Plugins with Static Libraries.

Description

A `ws` (WebSocket) endpoint supports all `tcp` endpoint options in addition to the following:

Option	Description	Client Semantics	Server Semantics
`-r` *resource*	A URI specifying the resource associated with this endpoint. If not specified, the default value is `/`.	The value for *resource* is passed as the target for `GET` in the WebSocket upgrade request.	The web server configuration must direct the given *resource* to this endpoint.

WSS Endpoint Syntax

Synopsis

```
wss -r resource -h host -p port -t timeout -z --sourceAddress addr
```

Availability

The WSS transport is a built-in transport and only enabled if an SSL transport is configured with the communicator. Like for the WS transport, the WS plug-in needs to be registered explicitly for C++ and Objective-C static builds. See Using Plugins with Static Libraries.

Description

A `wss` (Secure WebSocket) endpoint supports all `ssl` endpoint options in addition to the following:

Option	Description	Client Semantics	Server Semantics
`-r` *resource*	A URI specifying the resource associated with this endpoint. If not specified, the default value is `/`.	The value for *resource* is passed as the target for `GET` in the WebSocket upgrade request.	The web server configuration must direct the given *resource* to this endpoint.

Bluetooth Endpoint Syntax

Synopsis

```
bt -a addr -u uuid -c channel -t timeout -z --name name
bts -a addr -u uuid -c channel -t timeout -z --name name
```

Availability

The Bluetooth transport is provided by a separate IceBT plug-in.

Description

Support for Bluetooth endpoints is provided by the IceBT transport plug-in. The plug-in enables the `bt` protocol by default; if the IceSSL transport plug-in is also installed, IceBT enables the `bts` protocol as well. A Bluetooth endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
`-a` *addr*	Specifies the Bluetooth device address in the form *xx:xx:xx:xx:xx:xx*. The address must be enclosed in quotes because the colon (`:`) character is also a separator in the endpoint syntax. If not specified, the value of `Ice.Default.Host` is used instead.	An address is required for proxy endpoints. The plug-in will throw `EndpointParseException` if no address is specified via the `-a` option or `Ice.Default.Host`.	On Linux, the plug-in uses the address to select the Bluetooth adapter on which to listen for new connections. If no address is specified via the `-a` option or `Ice.Default.Host`, the plug-in uses the system's default adapter. On Android, the plug-in ignores this setting and always uses the system's default adapter. In addition to a Bluetooth device address, *addr* can also be `*` (e.g., `bt -a * ...`) to force the endpoint to use the system's default adapter without having to specify its address directly. The plug-in ignores the setting for `Ice.Default.Host` in this case.
`-u` *uuid*	Specifies the UUID of a service.	A UUID is required for proxy endpoints. The plug-in uses the UUID to search for a matching service at the specified device address.	A UUID is optional (but recommended) for object adapter endpoints. If this option is not specified, the plug-in generates a random UUID. The plug-in registers the service with its UUID in the local SDP registry so that clients can locate it.
`-c` *channel*	Specifies the RFCOMM channel number of the endpoint.	Not supported in proxy endpoints – the plug-in always uses the service UUID to connect to a server.	This setting is optional on Linux. The value for *channel* must be in the range 0-30. An available channel will be selected automatically if this setting is not specified or *channel* is zero. This setting is ignored on Android – the system always selects the channel.
`-t` *timeout*	Specifies the endpoint timeout in milliseconds.	The value for *timeout* must either be `infinite` to specify no timeout, or an integer greater than zero representing the timeout in milliseconds used by the client to open or close connections and to read or write data. If a timeout occurs, the application receives `Ice::TimeoutException`. If this option is not specified, it defaults to the value of `Ice.Default.Timeout`.	The value for *timeout* must either be `infinite` to specify no timeout, or an integer greater than zero representing the timeout in milliseconds used by the server to accept or close connections and to read or write data (see Timeouts in Object Adapter Endpoints and Connection Timeouts). *timeout* also controls the timeout that is published in proxies created by the object adapter. If this option is not specified, it defaults to the value of `Ice.Default.Timeout`.
`-z`	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.
`--name` *name*	Specifies the service name.	Ignored in proxy endpoints.	Associates a human-friendly name with the service's entry in the SDP registry. If not specified, the default name is `Ice Service`.

iAP Endpoint Syntax

Synopsis

```
iap -p protocol -n name -m manufacturer -o model number -t timeout -z
iaps -p protocol -n name -m manufacturer -o model number -t
```

Availability

The iAP transport is provided by a separate IceIAP plug-in.

Description

Support for iAP endpoints is provided by the IceIAP transport plug-in. The plug-in enables the `iap` protocol by default; if the IceSSL transport plug-in is also installed, IceIAP enables the `iaps` protocol as well. The iAP protocol allows applications running on iOS to communicate with accessories connected to the iOS device either through Bluetooth or the lightning connector. The iAP transport is a client side transport. There's no server-side support. An iAP endpoint supports the following options to allow selecting the accessory to connect to:

Option	Description	Client Semantics
`-p` *`protocol`*	Specifies the protocol implemented by the accessory.	When specified, the iAP transport will only connect to accessories that advertise the specified protocol.
`-n` *`name`*	Specifies the name of the accessory.	When specified, the iAP transport will only connect to accessories that match the specified accessory name.
`-m` *`manufacturer`*	Specifies the manufacturer of the accessory.	When specified, the iAP transport will only connect to accessories that match the specified accessory manufacturer.
`-o` *`model number`*	Specifies the model number of the accessory.	When specified, the iAP transport will only connect to accessories that match the specified accessory model number.
`-t` *`timeout`*	Specifies the endpoint timeout in milliseconds.	The value for *`timeout`* must either be `infinite` to specify no timeout, or an integer greater than zero representing the timeout in milliseconds used by the client to open or close connections and to read or write data. If a timeout occurs, the application receives `Ice::TimeoutException`. If this option is not specified, it defaults to the value of `Ice.Default.Timeout`.
`-z`	Specifies bzip2 compression.	Determines whether compressed requests are sent.

Opaque Endpoint Syntax

Synopsis

```
opaque -t type -e encoding -v value
```

Description

Proxies can contain endpoints that are not universally understood by Ice processes. For example, a proxy can contain an SSL endpoint; if that proxy is marshaled to a receiver without the IceSSL plug-in, the SSL endpoint does not make sense to the receiver.

Ice preserves such unknown endpoints when they are received over the wire. For the preceding example, if the receiver remarshals the proxy and sends it back to an Ice process that does have the IceSSL plug-in, that process can invoke on the proxy using its SSL transport. This mechanism allows proxies containing endpoints for arbitrary transports to pass through processes that do not understand these endpoints without losing information.

If an Ice process stringifies a proxy containing an unknown endpoint, it writes the endpoint as an opaque endpoint. For example:

```
opaque -t 2 -e 1.0 -v CTEyNy4wLjAuMREnAAD////AA==
```

This is how a process without the IceSSL plug-in stringifies an SSL endpoint. When a process with the IceSSL plug-in unstringifies this endpoint and converts it back into a string, it produces:

```
ssl -h 127.0.0.1 -p 10001
```

An `opaque` endpoint supports the following options:

Option	Description
`-t` *`type`*	Specifies the transport for the endpoint. Transports are indicated by positive integers (1 for TCP, 2 for SSL, and 3 for UDP).
`-e` *`encoding`*	Specifies the encoding version used to encode the endpoint marshaled data.
`-v` *`value`*	Specifies the marshaled encoding of the endpoint in base-64 encoding.

Exactly one each of the `-t` and `-v` options must be present in an opaque endpoint. If `-e` is not specified, the current encoding supported by the Ice runtime is assumed.

See Also

- The Ice Protocol
- Object Adapter Endpoints
- Connection Timeouts
- Ice.Default.*

Request Contexts

Methods on a proxy are overloaded with a trailing parameter representing a *request context*. The Slice definition of this parameter is as follows:

Slice

```
module Ice
{
    dictionary<string, string> Context;
}
```

As you can see, a context is a dictionary that maps strings to strings or, conceptually, a context is a collection of name-value pairs. The contents of this dictionary (if any) are implicitly marshaled with every request to the server, that is, if the client populates a context with a number of name-value pairs and uses that context for an invocation, the name-value pairs that are sent by the client are available to the server.

On the server side, the operation implementation can access the received `Context` via the `ctx` member of the `Ice::Current` parameter and extract the name-value pairs that were sent by the client.

Context names beginning with an underscore are reserved for use by Ice.

Topics

See Also

- The Current Object

Explicit Request Contexts

Request contexts provide a means of sending an unlimited number of parameters from client to server without having to mention these parameters in the signature of an operation. For example, consider the following definition:

Slice

```
struct Address
{
    // ...
}

interface Person
{
    string setAddress(Address a);
    // ...
}
```

Assuming that the client has a proxy to a `Person` object, it could do something along the following lines:

C++11C++98C#JavaJava CompatMATLAB

```
shared_ptr<PersonPrx> p = ...;
Address a = ...;

Ice::Context ctx;
ctx["write policy"] = "immediate";

p->setAddress(a, ctx);
```

```
PersonPrx p = ...;
Address a = ...;

Ice::Context ctx;
ctx["write policy"] = "immediate";

p->setAddress(a, ctx);
```

```
using System.Collections.Generic;

PersonPrx p = ...;
Address a = ...;

Dictionary<string, string> ctx = new Dictionary<string, string>();
ctx["write policy"] = "immediate";

p.setAddress(a, ctx);
```

```
PersonPrx p = ...;
Address a = ...;

java.util.Map<String, String> ctx = new java.util.HashMap<String,
String>();
ctx.put("write policy", "immediate");

p.setAddress(a, ctx);
```

```
PersonPrx p = ...;
Address a = ...;

java.util.Map<String, String> ctx = new java.util.HashMap<String,
String>();
ctx.put("write policy", "immediate");

p.setAddress(a, ctx);
```

```
person = ...;
addr = ...;

ctx = containers.Map('KeyType', 'char', 'ValueType', 'char');
ctx('write policy') = 'immediate';

person.setAddress(addr, ctx);
```

On the server side, we can extract the policy value set from the `Current` object to influence how the implementation of `setAddress` works. For example:

C++11 C++98 C# Java Java Compat

```
void
PersonI::setAddress(Address a, const Ice::Current& current)
{
    auto i = c.ctx.find("write policy");
    if(i != c.ctx.end() && i->second == "immediate")
    {
        // Update the address details and write through to the
        // data base immediately...
    }
    else
    {
        // Write policy was not set (or had a bad value), use
        // some other database write strategy.
    }
}
```

```
void
PersonI::setAddress(const Address& a, const Ice::Current& current)
{
    Ice::Context::const_iterator i = c.ctx.find("write policy");
    if(i != c.ctx.end() && i->second == "immediate")
    {
        // Update the address details and write through to the
        // data base immediately...
    }
    else
    {
        // Write policy was not set (or had a bad value), use
        // some other database write strategy.
    }
}
```

```
void setAddress(Address a, Ice.Current current)
{
    if(current.ctx.ContainsKey("write policy") && current.ctx["write
policy"] == "immediate")
    {
         // Update the address details and write through to the
         // data base immediately...
    }
    else
    {
        // Write policy was not set (or had a bad value), use
        // some other database write strategy.
    }
}
```

```
@Override
void setAddress(Address a, com.zeroc.Ice.Current current)
{
    String writePolicy = current.ctx.get("write policy);
    if(writePolicy != null && writePolicy.equals("immediate"))
    {
         // Update the address details and write through to the
         // data base immediately...
    }
    else
    {
        // Write policy was not set (or had a bad value), use
        // some other database write strategy.
    }
}
```

```
@Override
void setAddress(Address a, Ice.Current current)
{
    String writePolicy = current.ctx.get("write policy);
    if(writePolicy != null && writePolicy.equals("immediate"))
    {
         // Update the address details and write through to the
         // data base immediately...
    }
    else
    {
        // Write policy was not set (or had a bad value), use
        // some other database write strategy.
    }
}
```

For this example, the server examines the value of the context with the key `"write policy"` and, if that value is `"immediate"`, writes the update sent by the client straight away; if the write policy is not set or contains a value that is not recognized, the server presumably applies a more lenient write policy (such as caching the update in memory and writing it later).

See Also

- Request Contexts
- The Current Object

Per-Proxy Request Contexts

Instead of passing a context explicitly with an invocation, you can also use a *per-proxy context*. Per-proxy contexts allow you to set a context on a particular proxy once and, thereafter, whenever you use that proxy to invoke an operation, the previously-set context is sent with each invocation.

On this page:

Configuring a Per-Proxy Request Context Programmatically

The proxy methods `ice_context` and `ice_getContext` set and retrieve the context, respectively. The Slice definitions of these methods would look as follows:

Slice

```
Object* ice_context(Context ctx);
Context ice_getContext();
```

`ice_context` creates a new proxy that stores the given context. Calling `ice_getContext` returns the stored context, or an empty dictionary if no per-proxy context has been configured for the proxy.

Here is an example in C++:

C++11 C++98

```
Ice::Context ctx;
ctx["write policy"] = "immediate";

shared_ptr<PersonPrx> p1 = ...;
auto p2 = p1->ice_context(ctx);

Address a = ...;

p1->setAddress(a);       // Sends no context

p2->setAddress(a);       // Sends ctx implicitly

Ice::Context ctx2;
ctx2["write policy"] = "delayed";

p2->setAddress(a, ctx2); // Sends ctx2 instead
```

```
Ice::Context ctx;
ctx["write policy"] = "immediate";

PersonPrx p1 = ...;
PersonPrx p2 = p1->ice_context(ctx);

Address a = ...;

p1->setAddress(a);       // Sends no context

p2->setAddress(a);       // Sends ctx implicitly

Ice::Context ctx2;
ctx2["write policy"] = "delayed";

p2->setAddress(a, ctx2); // Sends ctx2 instead
```

As the example illustrates, once we have created the `p2` proxy, any invocation via `p2` automatically sends the configured context. The final line of the example shows that it is also possible to explicitly send a context for an invocation even if the proxy is configured with a context — an explicit context always overrides any per-proxy context.

Note that, once you have set a per-proxy context, that context becomes immutable: if you subsequently change the context you have passed to `ice_context`, such a change does not affect the per-proxy context of any proxies you previously created with that context because each proxy on which you set a per-proxy context stores its own copy of the dictionary.

Configuring a Per-Proxy Request Context using Properties

You can also configure a context with proxy properties when you use the communicator operation `propertyToProxy`. Suppose we obtain a proxy like this:

C++11

```
auto communicator = ...;
auto proxy = communicator->propertyToProxy("PersonProxy");
```

We can statically configure a context for this proxy using the following properties:

```
PersonProxy=person:tcp -p 5000
PersonProxy.Context.write policy=immediate
PersonProxy.Context.failure mode=persistent
```

The `Context` property has the form *`name`*`.Context.`*`key`*`=`*`value`*, where *`key`* and *`value`* can be any legal property symbols.

The proxy returned by `propertyToProxy` already contains the context key/value pairs specified in the configuration properties. To make any modifications to the context at run time, you'll need to retrieve the proxy's context dictionary using `ice_getContext`, modify the dictionary as necessary, and finally obtain a new proxy by calling `ice_context`, as we described above.

See Also

- Explicit Request Contexts
- Proxy Methods

- Proxy Properties

Implicit Request Contexts

On this page:

Using Implicit Request Contexts

In addition to explicit and per-proxy request contexts, you can also establish an implicit context on a communicator. This implicit context is sent with all invocations made via proxies created by that communicator, provided that you do not supply an explicit context with the call.

Access to this implicit context is provided by the `Communicator` interface:

Slice

```
module Ice
{
    local interface Communicator
    {
        ImplicitContext getImplicitContext();
        // ...
    }
}
```

`getImplicitContext` returns the implicit context object. If a communicator has no implicit context, the operation returns null.

You can manipulate the contents of the implicit context via the `ImplicitContext` interface:

Slice

```
local interface ImplicitContext
{
    Context getContext();
    void    setContext(Context newContext);

    string get(string key);
    string put(string key, string value);
    string remove(string key);
    bool   containsKey(string key);
}
```

The `getContext` operation returns the currently-set context dictionary. The `setContext` operation replaces the currently-set context in its entirety.

The remaining operations allow you to manipulate specific entries:

- `get`
  This operation returns the value associated with `key`. If `key` was not previously set, the operation returns the empty string.
- `put`
  This operation adds the key-value pair specified by `key` and `value`. It returns the previous value associated with `key`; if no value was previously associated with `key`, it returns the empty string. It is legal to add the empty string as a value.
- `remove`

This operation removes the key-value pair specified by `key`. It returns the previously-set value (or the empty string if `key` was not previously set).

- `containsKey`
  This operation returns true if `key` is currently set and false, otherwise. You can use this operation to distinguish between a key-value pair that was explicitly added with an empty string as a value, and a key-value pair that was never added at all.

Scope of the Implicit Context

You establish the implicit context on a communicator by setting a property, `Ice.ImplicitContext`. This property controls whether a communicator has an implicit context and, if so, at what scope the context applies. The property can be set to the following values:

- `None`
  With this setting (or if `Ice.ImplicitContext` is not set at all), the communicator has no implicit context, and `getImplicitContext` returns null.
- `Shared`
  The communicator has a single implicit context that is shared by all threads. Access to the context via its `ImplicitContext` interface is interlocked, so different threads can concurrently manipulate the context without risking data corruption or reading stale values.
- `PerThread`
  The communicator maintains a separate implicit context for each thread. This allows you to propagate contexts that depend on the sending thread (for example, to send per-thread transaction IDs).

See Also

- Explicit Request Contexts
- Per-Proxy Request Contexts

Design Considerations for Request Contexts

On this page:

Request Context Interactions

If you use explicit, per-proxy, and implicit contexts, it is important to be aware of their interactions:

- If you send an explicit context with an invocation, *only* that context is sent with the call, regardless of whether the proxy has a per-proxy context and whether the communicator has an implicit context.
- If you send an invocation via a proxy that has a per-proxy context, and the communicator also has an implicit context, the contents of the per-proxy and implicit context dictionaries are combined, so the *combination* of context entries of both contexts is transmitted to the server. If the per-proxy context and the implicit context contain the same key, but with different values, the *per-proxy* value takes precedence.

Request Context Use Cases

The purpose of `Ice::Context` is to permit services to be added to Ice that require some contextual information with every request. Contextual information can be used by services such as a transaction service (to provide the context of a currently established transaction) or a security service (to provide an authorization token to the server). IceStorm uses the context to provide an optional `cost` parameter to the service that influences how the service propagates messages to down-stream subscribers, and Glacier2 uses the context to influence request routing.

In general, services that require such contextual information can be implemented much more elegantly using contexts because this hides explicit Slice parameters that would otherwise have to be supplied by the application programmer with every call. For a request-forwarding intermediary service such as IceStorm or Glacier2, using a parameter is not an option because the service does not know the signatures of the requests that it is forwarding, but the service does have access to the context. For this use case, the context is a convenient way for the caller to annotate a request.

In addition, contexts, because they are optional, permit a single Slice definition to apply to implementations that use the context, as well as to implementations that do not use it. In this way, to add transactional semantics to an existing service, you do not need to modify the Slice definitions to add an extra parameter to each operation. (Adding an extra parameter would not only be inconvenient for clients, but would also split the type system into two halves: without contexts, we would need different Slice definitions for transactional and non-transactional implementations of (conceptually) a single service.)

Finally, per-proxy contexts permit context information to be passed through intermediate parts of your program without cooperation of those intermediate parts. For example, suppose you set a per-proxy context on a proxy and then pass that proxy to another function. When that function uses the proxy to invoke an operation, the per-proxy context will still be sent. In other words, per-proxy contexts allow you to transparently propagate information via intermediaries that are ignorant of the presence of any context.

Keep in mind though that this works only within a single process. If you stringify a proxy or transmit it as a parameter over the wire, the per-proxy context is *not* preserved. (Ice does not write the per-proxy context into stringified proxies and does not marshal the per-proxy context when a proxy is marshaled.)

Recommendations for Request Contexts

Contexts are a powerful mechanism for transparent propagation of context information, *if used correctly*. In particular, you may be tempted to use contexts as a means of versioning an application as it evolves over time. For example, version 2 of your application may accept two parameters on an operation that, in version 1, used to accept only a single parameter. Using contexts, you could supply the second parameter as a name-value pair to the server and avoid changing the Slice definition of the operation in order to maintain backward compatibility.

We *strongly* urge you to resist any temptation to use contexts in this manner. The strategy is fraught with problems:

- Missing context
  There is nothing that would compel a client to actually send a context when the server expects to receive a context: if a client forgets to send a context, the server, somehow, has to make do without it (or throw an exception).

- Missing or incorrect keys
  Even if the client does send a context, there is no guarantee that it has set the correct key. (For example, a simple spelling error can cause the client to send a value with the wrong key.)

- Incorrect values
  The value of a context is a string, but the application data that is to be sent might be a number, or it might be something more complex, such as a structure with several members. This forces you to encode the value into a string and decode the value again on the server side. Such parsing is tedious and error prone, and far less efficient than sending strongly-typed parameters. In addition, the server has to deal with string values that fail to decode correctly (for example, because of an encoding error made by the client).

None of the preceding problems can arise if you use proper Slice parameters: parameters cannot be accidentally omitted and they are strongly typed, making it much less likely for the client to accidentally send a meaningless value. Furthermore, you can use optional parameters to modify the signature of an operation without breaking backward compatibility.

Contexts are meant to be used to transmit simple tokens (such as a transaction identifier) for services that cannot be reasonably implemented without them; you should restrict your use of contexts to that purpose and resist any temptation to use contexts for any other purpose.

Finally, be aware that, if a request is routed via one or more Ice routers, contexts may be dropped by intermediate routers if they consider them illegal. This means that, in general, you cannot rely on an arbitrary context value that is created by an application to actually still be present when a request arrives at the server — only those context values that are known to routers and that are considered legitimate are passed on. It follows that you should not abuse contexts to pass things that really should be passed as parameters.

See Also

- Explicit Request Contexts
- Per-Proxy Request Contexts
- Implicit Request Contexts
- Optional Values
- Versioning
- IceStorm

Invocation Timeouts

On this page:

Overview of Invocation Timeouts

Invocation timeouts let an application specify the maximum amount of time it's willing to wait for invocations to complete. If the timeout expires, the application receives `InvocationTimeoutException` as the result of an invocation. The Ice runtime starts the timer for the invocation timeout after the marshalling of the invocation input parameters and before its starts any network activity (connection establishment and sending of the request over the network connection). For a twoway invocation, it stops the timer as soon as the response is received from the server and before the invocation output parameters are un-marshalled. For a oneway invocation, it stops the timer as soon as the invocation is sent.

> Use connection timeouts to detect network failures in a reasonable period of time.

> Invocation timeouts were introduced in Ice 3.6. In earlier Ice versions, connection timeouts were also used as invocation timeouts.

Configuring the Default Invocation Timeout

The property `Ice.Default.InvocationTimeout` establishes the default timeout value for invocations. This property has a default value of `-1`, which means invocations do not time out by default. If defined to `-2`, invocation timeouts are disabled and the Ice run time behaves like Ice versions < 3.6: it uses connection timeouts (defined on the endpoints) to wait for the response of the invocation. This setting is provided only for backward compatibility and might be deprecated in a future Ice major release.

Consider this setting:

```
Ice.Default.InvocationTimeout=5000
```

This configuration causes all invocations to time out if they do not complete within five seconds. Generally speaking however, it's unlikely that a single timeout value will be appropriate for all of the operations that an application invokes. It's more common for applications to configure invocation timeouts on a per-proxy basis, as we describe in the next section.

Configuring Invocation Timeouts for Proxies

You have a couple of options for configuring invocation timeouts at the proxy level:

- Use a proxy property
- Call `ice_invocationTimeout`

Assuming you've defined a configuration property containing a proxy that your application reads using `propertyToProxy`, you can statically configure an invocation timeout as follows:

```
# Assumes the application calls propertyToProxy("MyProxy")
MyProxy=theIdentity:tcp -p 5000
MyProxy.InvocationTimeout=2500  # 2.5 seconds
```

The `InvocationTimeout` proxy property specifies the invocation timeout that will be used for all invocations made via the proxy returned by `propertyToProxy`.

To configure an invocation timeout at run time, use the `ice_invocationTimeout` factory method to obtain a new proxy with the desired timeout:

C++11 C++98

```
shared_ptr<Filesystem::FilePrx> myFile = ...;
auto timeoutFile = myFile->ice_invocationTimeout(2500);  // 2.5 seconds
```

```
Filesystem::FilePrx myFile = ...;
FileSystem::FilePrx timeoutFile = myFile->ice_invocationTimeout(2500);
// 2.5 seconds
```

Invocation Timeout Failures

An application that configures invocation timeouts must be prepared to catch `InvocationTimeoutException`:

C++11 C++98

```
shared_ptr<Filesystem::FilePrx> myFile = ...;
auto timeoutFile = myFile->ice_invocationTimeout(2500);

try
{
    auto text = timeoutFile->read();   // Read with timeout
}
catch(const Ice::InvocationTimeoutException&)
{
    cerr << "invocation timed out" << endl;
}

auto text = myFile->read();            // Read without timeout
```

```
Filesystem::FilePrx myFile = ...;
FileSystem::FilePrx timeoutFile = myFile->ice_invocationTimeout(2500);

try
{
    Lines text = timeoutFile->read();   // Read with timeout
}
catch(const Ice::InvocationTimeoutException&)
{
    cerr << "invocation timed out" << endl;
}

Lines text = myFile->read();            // Read without timeout
```

The effects of an invocation timeout are limited to the client; no indication is sent to the server, which may still be busy dispatching the request. The Ice run time in the client ignores a reply for this request if the server eventually sends one.

> Ice does **not** perform automatic retries for invocation timeouts.

See Also

- Proxy Methods
- Proxy Properties
- Ice.Default.*
- Connection Timeouts
- Obtaining Proxies

Automatic Retries

Ice may automatically retry a proxy invocation after a failure. This is a powerful feature that, when used in the proper situations, can significantly improve the robustness of your application without any additional programming effort. The retry facility is governed by one overriding principle: always respect at-most-once semantics. At-most-once semantics dictate that the Ice run time in the client must never retry a failed proxy invocation unless Ice guarantees that the server has not already received the request, or unless the application declares that it is safe for Ice to violate at-most-once semantics for the request.

To understand the importance of obeying at-most-once semantics, consider the following Slice definition:

Slice

```
interface Account
{
    long withdraw(long amount);
}
```

The `withdraw` operation removes funds from an account. If an invocation of `withdraw` fails, automatically retrying the request introduces the risk of a duplicate withdrawal unless Ice is absolutely sure that the server has not already executed the request.

This page examines automatic retries in more detail.

On this page:

Automatic Retries for Request Failures

Ice considers a request to have failed if any of the following conditions are true:

- A connection could not be established
- A connection was lost before the reply was received
- A timeout expired
- An exception occurred while sending the request or receiving the reply
- An error occurred in the server while dispatching the request that causes the server to return an `UnknownException` or `RequestFailedException`

> Ice considers an invocation that results in a user exception to be successful and therefore excludes it from consideration for automatic retries.

Ice must determine the answers to several questions to decide whether to retry a failed request:

1. What kind of error caused the request to fail?

Ice does not bother retrying a request if it knows the same error is going to occur again. For example, Ice never retries an invocation that raises a `MarshalException`, which indicates that there was a problem while encoding or decoding a message. Retrying such an invocation is unlikely to change the outcome. It also doesn't retry invocation timeouts, if a server didn't respond within the invocation timeout period, it's unlikely that a retry would provide better results.

Ice also never retries exceptions that derive from `RequestFailedException` because they indicate a permanent failure. One such subclass is `OperationNotExistException`, whose occurrence signals a serious problem in the application. For instance, it might mean

that the client and server are using incompatible Slice definitions, or that the client is trying to invoke operations on the wrong object. The exception to this rule is `ObjectNotExistException`, which Ice does consider to be worthy of retry if the proxy in question is indirect because it gives an application the ability to transparently migrate an Ice object.

In addition to user exceptions and subclasses of `RequestFailedException`, a server can also return an instance of `UnknownException`, `UnknownLocalException`, or `UnknownUserException` to indicate that it encountered an unexpected exception while dispatching the request. These exceptions *are* eligible for retry.

2. When did the error occur?

If the error is still a candidate for retry, Ice needs to know whether the server has received the request. Naturally, the Ice run time in the client cannot possibly know that information until the server confirms it by sending a reply. However, to be conservative Ice assumes that the server has received the request as soon as Ice has written the *entire* protocol message to the client's local transport buffers. If the error occurred before Ice managed to write the complete message, retrying the request would not violate at-most-once semantics.

The Ice run time in the server also has the ability to notify the client that a request was not dispatched and therefore that it is safe for the Ice run time in the client to retry the request without violating at-most-once semantics. For example, this situation can occur when the server is shutting down while there are pending requests that have yet to be executed. Sending this notification allows a client to transparently fail over to another server.

3. Does the application require strict adherence to at-most-once semantics for this request?

An application can grant permission for Ice to violate at-most-once semantics for certain Slice operations by marking them as *idempotent*, causing Ice to retry a request that otherwise would be ineligible because the server has already received it. We discuss idempotent operations in more detail below.

If Ice determines that an invocation cannot be retried, it raises the exception that caused the request failure to the application. On the other hand, if Ice does retry the invocation and the subsequent retries also fail, Ice raises the *last* exception to the application. For example, if the first attempt fails with `ConnectionRefusedException` and the retry fails with `ConnectTimeoutException`, the invocation raises `ConnectTimeoutException` to the application.

Automatic Retries for Idempotent Operations

Annotating a Slice operation with the `idempotent` keyword notifies Ice that it can safely violate at-most-once semantics:

Slice

```
interface Account
{
    long withdraw(long amount);
    idempotent long getBalance();
}
```

Although `withdraw` clearly requires the stricter treatment, there is no harm in automatically retrying the `getBalance` operation even if the server executes the same request more than once.

In general, "read-only" operations are good candidates for the `idempotent` keyword whereas many mutating operations are not. However, the risk of duplicate requests is acceptable even for some kinds of mutating operations:

Slice

```
interface Account
{
    long withdraw(long amount);
    idempotent long getBalance();
    idempotent void changeAddress(string newAddress);
}
```

Here we have marked `changeAddress` as idempotent because executing the request twice has the same effect as executing it only once.

The benefit of the `idempotent` keyword and the associated relaxation of retry semantics is that an invocation that otherwise might have raised an exception has at least one more chance to succeed. Furthermore, the application does not need to initiate the retry, and in fact the retry activities are completely transparent: if a subsequent retry succeeds, the application receives its results as if nothing went wrong. The invocation only raises an exception once Ice has reached its configured retry limits.

Configuring Automatic Retries

Retry Intervals

The `Ice.RetryIntervals` property configures the retry behavior for a communicator and affects invocations on every proxy created by that communicator. (Retry behavior cannot be configured on a per-proxy basis.) The value of this property consists of a series of integers separated by whitespace. The number of integers determines how many retry attempts Ice makes, and the value of each entry represents a delay in milliseconds. If this property is not defined, the default behavior is to retry once immediately after the first failure, which is equivalent to the following property definition:

```
Ice.RetryIntervals=0
```

You may want a more elaborate configuration for your application, such as a gradual increase in the delay between retries:

```
Ice.RetryIntervals=0 100 500 1000
```

With this setting, Ice retries immediately as in the default case. If the first retry attempt also fails, Ice waits 100 milliseconds before trying again, then 500 milliseconds, and finally tries one more time after waiting one second.

In some situations you may need to disable retries completely. For example, an application might implement its own retry logic and therefore require immediate notification when a failure occurs. Clients that establish a session with a Glacier2 router also need to disable retries. To prevent automatic retries, use a value of `-1`:

```
Ice.RetryIntervals=-1
```

Retry Logging

To monitor Ice's retry activities, configure your program with the property `Ice.Trace.Retry` set to a non-zero value:

```
Ice.Trace.Retry=1
```

When retry tracing is enabled, Ice logs a message each time it attempts a retry; the log message includes a description of the exception that prompted the retry. Ice also logs a message when it reaches the retry limit.

You can configure Ice to log even more information about retries by setting the property to `2`:

```
Ice.Trace.Retry=2
```

This setting prompts Ice to include additional details about connections and endpoints.

Timeouts and Automatic Retries

Ice does not retry an invocation that fails with an invocation timeout. However, an invocation that fails with a connection timeout can be

eligible for retry.

If you test connection timeouts (for example, by attempting to connect to an unreachable IP address), you may notice that the `TimeoutException` is not raised as quickly as you would expect . Automatic retries are usually the reason for this situation.

For example, suppose a proxy is configured with a ten seconds connection timeout and automatic retries are enabled with the default setting (one immediate retry). If an invocation on that proxy fails due to a connection timeout and Ice determines that the invocation is eligible for retry (using the criteria described above), Ice immediately tries the invocation again and waits for another connection timeout period to expire before finally raising `TimeoutException`. From the application's perspective, the invocation fails after approximately twenty seconds.

Consequently, you can compute an approximate worst-case connection timeout value as follows, assuming the proxy has a single endpoint:

```
T = t * (N + 1) + D
```

where *t* is the connection timeout value, *N* is the number of retry intervals, and *D* is the sum of the retry intervals (the total delay between retries). Consider our example again:

```
Ice.RetryIntervals=0 10000 20000 30000
```

Using this configuration with a ten second connection timeout, our approximate worst-case timeout is `10 * 5 + 60 = 110` seconds.

Connections and Automatic Retries

The behavior of automatic retries is intimately tied to the presence (and absence) of connections. This section describes the errors that cause Ice to close connections, and provides more details about how connections influence retries.

Connection Errors

Ice automatically closes a connection in response to certain fatal error conditions. Of these, the one that is the most likely to affect Ice applications is a connection timeout. Other errors that prompt Ice to close a connection include the following:

- a socket failure while performing I/O on the connection
- receiving an improperly formatted message
- dispatching an operation to a Java servant raises `OutOfMemoryError` or `AssertionError`

When Ice closes a connection in response to one of these errors, all other outstanding requests on the same connection also fail and may be retried if eligible.

Connection Status

One factor that influences retry behavior is the status of the connection on which the failed request was attempted. If the failure caused Ice to close the connection (as discussed in the previous section), or if the request failed because Ice could not establish a connection, Ice must try to obtain another connection before it can retry the request.

It is also important to understand that Ice may not retry the invocation on the original endpoint *even if the connection that was used for the initial request remains open*. The retry behavior in this case depends on several criteria:

- whether the proxy caches its connection
- whether the proxy contains multiple endpoints
- whether other connections exist to any of the proxy's endpoints
- the proxy's configured endpoint selection type

Generally speaking, you must configure your application carefully if you need fine-grained control over Ice's retry behavior.

Automatic Retries: Direct Proxy versus Indirect Proxies

With a direct proxy, Ice tries to establish a connection using each suitable endpoint of the proxy, and, if this fails, Ice retries these connection attempts (Ice retries once immediately with the default retry configuration).

With an indirect proxy, the retry algorithm is a little bit different:

- Ice first attempts to establish a connection using the endpoints found in its locator cache, with one attempt for each suitable endpoint.
- if this fails, Ice refreshes its locator cache and tries to establish a connection using the refreshed endpoints (this new attempt with just-refreshed endpoints does not count as a retry).
- if all these attempts still fail, Ice refreshes its locator cache again and tries to establish a connection to the re-refreshed endpoints, which represents retry attempt number 1

See Also

- Terminology
- Operations
- Invocation Timeouts
- Connection Timeouts
- Connection Establishment
- Getting Started with Glacier2

Oneway Invocations

On this page:

Design Considerations for Oneway Invocations

A oneway invocation is sent on the client side by writing the request to the client's local transport buffers; the invocation completes and returns control to the application code as soon as it has been accepted by the local transport. Of course, this means that a oneway invocation is unreliable: it may never be sent (for example, because of a network failure) or it may not be accepted in the server (for example, because the target object does not exist).

This is an issue in particular if you use Active Connection Management (ACM): if a server closes a connection at the wrong moment, it's possible for the client to lose already-buffered oneway requests. If your client uses oneway (or batched oneway) requests, we recommend that you either disable ACM for the server side, or enable ACM heartbeats in the client to ensure the connection remains active. In addition, if clients use oneway requests and your application initiates server shutdown, it is the responsibility of your application to ensure either that it can cope with the potential loss of buffered oneway requests, or that it does not shut down the server at the wrong moment (while clients still have oneway requests that are buffered, but not yet sent).

If anything goes wrong with a oneway request, the client-side application code does not receive any notification of the failure; the only errors that are reported to the client are local errors that occur on the client side during call invocation (such as failure to establish a connection, for example).

As a consequence of oneway invocation, if you call `ice_ping` on a oneway proxy, successful completion does *not* indicate that the target object exists and could successfully be contacted — you will receive an exception only if something goes wrong on the client side, but not if something goes wrong on the server side. Therefore, if you want to use `ice_ping` with a oneway proxy and be certain that the target object exists and can successfully be contacted, you must first convert the oneway proxy into a twoway proxy. For example, in C++:

C++11 C++98

```
shared_ptr<SomeObjectPrx> onewayPrx = ...; // Get a oneway proxy

try
{
    onewayPrx->ice_twoway()->ice_ping();
}
catch(const Ice::Exception&)
{
    cerr << "object not reachable" << endl;
}
```

```
SomeObjectPrx onewayPrx = ...; // Get a oneway proxy

try
{
    onewayPrx->ice_twoway()->ice_ping();
}
catch(const Ice::Exception&)
{
    cerr << "object not reachable" << endl;
}
```

Oneway invocations are received and processed on the server side like any other incoming request. If necessary, a server can distinguish a

oneway invocation by examining the `requestId` member of `Ice::Current`: a non-zero value denotes a twoway request, whereas a value of zero indicates a oneway request.

Oneway invocations do not incur any return traffic from the server to the client: the server never sends a reply message in response to a oneway invocation. This means that oneway invocations can result in large efficiency gains, especially for large numbers of small messages, because the client does not have to wait for the reply to each message to arrive before it can send the next message.

In order to be able to invoke an operation as oneway, two conditions must be met:

- The operation must have a `void` return type, must not have any out-parameters, and must not have an exception specification.

  This requirement reflects the fact that the server does not send a reply for a oneway invocation to the client: without such a reply, there is no way to return any values or exceptions to the client. If you attempt to invoke an operation that returns values to the client as a oneway operation, the Ice run time throws a `TwowayOnlyException` (for asynchronous invocations with the C++98 and Java-Compat mappings, `IceUtil::IllegalArgumentException` and `java.lang.IllegalArgumentException` are raised instead respectively).

- The proxy on which the operation is invoked must support a stream-oriented transport (such as TCP or SSL).

  Oneway invocations require a stream-oriented transport. (To get something like a oneway invocation for datagram transports, you need to use a datagram invocation.) If you attempt to create a oneway proxy for an object that does not offer a stream-oriented transport, the Ice run time throws a `NoEndpointException`.

Despite their theoretical unreliablity, in practice, oneway invocations are reliable (but not infallible [1]): they are sent via a stream-oriented transport, so they cannot get lost except when the connection is shutting down or fails entirely. In particular, the transport uses its usual flow control, so the client cannot overrun the server with messages. On the client-side, the Ice run time will block if the client's transport buffers fill up, so the client-side application code cannot overrun its local transport.

Consequently, oneway invocations normally do not block the client-side application code and return immediately, provided that the client does not consistently generate messages faster than the server can process them. If the rate at which the client invokes operations exceeds the rate at which the server can process them, the client-side application code will eventually block in an operation invocation until sufficient room is available in the client's transport buffers to accept the invocation. If your application requires that oneway requests never block the calling thread, you can use asynchronous oneway invocations instead.

Regardless of whether the client exceeds the rate at which the server can process incoming oneway invocations, the execution of oneway invocations in the server proceeds asynchronously: the client's invocation completes before the message even arrives at the server.

One thing you need to keep in mind about oneway invocations is that they may appear to be reordered in the server: because oneway invocations are sent via a stream-oriented transport, they are guaranteed to be received in the order in which they were sent. However, the server's thread pool may dispatch each invocation in its own thread; because threads are scheduled preemptively, this may cause an invocation sent later by the client to be dispatched and executed before an invocation that was sent earlier. If oneway requests must be dispatched in order, you can use one of the serialization techniques described in Thread Pool Design Considerations.

For these reasons, oneway invocations are usually best suited to simple updates that are otherwise stateless (that is, do not depend on the surrounding context or the state established by previous invocations).

Creating Oneway Proxies

Ice selects between twoway, oneway, and datagram invocations via the proxy that is used to invoke the operation. By default, all proxies are created as twoway proxies. To invoke an operation as oneway, you must create a new proxy configured specifically for oneway invocations. The `ice_oneway` factory method is provided for this purpose. The Slice definition of `ice_oneway` would look as follows:

Slice

```
Object* ice_oneway();
```

We can call `ice_oneway` to create a oneway proxy and then use the proxy to invoke an operation as follows:

C++11 C++98

```
auto o = communicator->stringToProxy(/* ... */);

// Get a oneway proxy.
//
auto oneway = o->ice_oneway();

// Down-cast to actual type.
//
auto onewayPerson = Ice::uncheckedCast<PersonPrx>(oneway);

// Invoke an operation as oneway.
//
try
{
    onewayPerson->someOp();
}
catch(const Ice::TwowayOnlyException&)
{
    cerr << "someOp() is not oneway" << endl;
}
```

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a oneway proxy.
//
Ice::ObjectPrx oneway = o->ice_oneway();

// Down-cast to actual type.
//
PersonPrx onewayPerson = PersonPrx::uncheckedCast(oneway);

// Invoke an operation as oneway.
//
try
{
    onewayPerson->someOp();
}
catch(const Ice::TwowayOnlyException&)
{
    cerr << "someOp() is not oneway" << endl;
}
```

Note that we use an `uncheckedCast` to down-cast the proxy from `ObjectPrx` to `PersonPrx`: for a oneway proxy, we cannot use a `checkedCast` because a `checkedCast` requires a reply from the server but, of course, a oneway proxy does not permit that reply. If instead you want to use a safe down-cast, you can first down-cast the twoway proxy to the actual object type and then obtain the oneway proxy:

C++11 C++98

```
auto o = communicator->stringToProxy(/* ... */);

// Safe down-cast to actual type.
//
auto person = Ice::checkedCast<PersonPrx>(o);

if(person)
{
    // Get a oneway proxy.
    //
    auto onewayPerson = person->ice_oneway();

    // Invoke an operation as oneway.
    //
    try
    {
        onewayPerson->someOp();
    }
    catch(const Ice::TwowayOnlyException&)
    {
        cerr << "someOp() is not oneway" << endl;
    }
}
```

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Safe down-cast to actual type.
//
PersonPrx person = PersonPrx::checkedCast(o);

if(person)
{
    // Get a oneway proxy.
    //
    PersonPrx onewayPerson = person->ice_oneway();

    // Invoke an operation as oneway.
    //
    try
    {
        onewayPerson->someOp();
    }
    catch(const Ice::TwowayOnlyException&)
    {
        cerr << "someOp() is not oneway" << endl;
    }
}
```

Note that, while the second version of this code is somewhat safer (because it uses a safe down-cast), it is also slower (because the safe down-cast incurs the cost of an additional twoway message).

See Also

- Terminology
- Active Connection Management
- Batched Invocations
- The Current Object
- The Ice Threading Model
- Datagram Invocations
- The Ice Protocol

References

1. Snader, J. C. 2000. Effective TCP/IP Programming. Reading, MA: Addison-Wesley.

Datagram Invocations

On this page:

Design Considerations for Datagram Invocations

Datagram invocations are the equivalent of oneway invocations for datagram transports. As for oneway invocations, datagram invocations can be sent only for operations that have a `void` return type and do not have out-parameters or an exception specification. Attempts to use a datagram invocation with an operation that does not meet these criteria result in a `TwowayOnlyException`. In addition, datagram invocations can only be used if the proxy's endpoints include at least one UDP transport; otherwise, the Ice run time throws a `NoEndpointException`.

> The UDP transport is a built-in transport except when using C++ or Objective-C static library builds. In this case, you need to explicitly register the UDP plug-in for the transport to be available. See Using Plugins with Static Libraries for additional information.

The semantics of datagram invocations are similar to oneway invocations: no return traffic flows from the server to the client and proceed asynchronously with respect to the client; a datagram invocation completes as soon as the client's transport has accepted the invocation into its buffers. However, datagram invocations differ in one respect from oneway invocations in that datagram invocations optionally support multicast semantics. Furthermore, datagram invocations have additional error semantics:

- Individual invocations may be lost or received out of order.

  On the wire, datagram invocations are sent as true datagrams, that is, individual datagrams may be lost, or arrive at the server out of order. As a result, not only may operations be dispatched out of order, an individual invocation out of a series of invocations may be lost. (This cannot happen for oneway invocations because, if a connection fails, *all* invocations are lost once the connection breaks down.)

- UDP packets may be duplicated by the transport.

  Because of the nature of UDP routing, it is possible for datagrams to arrive in duplicate at the server. This means that, for datagram invocations, Ice does *not* guarantee at-most-once semantics: if UDP datagrams are duplicated, the same invocation may be dispatched more than once in the server.

- UDP packets are limited in size.

  The maximum size of an IP datagram is 65,535 bytes. Of that, the IP header consumes 20 bytes, and the UDP header consumes 8 bytes, leaving 65,507 bytes as the maximum payload. If the marshaled form of an invocation, including the Ice request header exceeds that size, the invocation is lost. (Exceeding the size limit for a UDP datagram is indicated to the application by a `DatagramLimitException`.)

Because of their unreliable nature, datagram invocations are best suited to simple update messages that are otherwise stateless. In addition, due to the high probability of loss of datagram invocations over wide area networks, you should restrict use of datagram invocations to local area networks, where they are less likely to be lost. (Of course, regardless of the probability of loss, you must design your application such that it can tolerate lost or duplicated messages.)

Creating Datagram Proxies

To invoke an operation as datagram, you must create a new proxy configured specifically for datagram invocations. The `ice_datagram` factory method is provided for this purpose. The Slice definition of `ice_datagram` would look as follows:

Slice

```
Object* ice_datagram();
```

We can call `ice_datagram` to create a oneway proxy and then use the proxy to invoke an operation as follows:

C++11 C++98

```
auto o = communicator->stringToProxy(/* ... */);

// Get a datagram proxy.
//
shared_ptr<Ice::ObjectPrx> datagram;
try
{
    datagram = o->ice_datagram();
}
catch(const Ice::NoEndpointException&)
{
    cerr << "No endpoint for datagram invocations" << endl;
}

// Down-cast to actual type.
//
auto datagramPerson = Ice::uncheckedCast<PersonPrx>(datagram);

// Invoke an operation as a datagram.
//
try
{
    datagramPerson->someOp();
}
catch(const Ice::TwowayOnlyException&)
{
    cerr << "someOp() is not oneway" << endl;
}
```

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a datagram proxy.
//
Ice::ObjectPrx datagram;
try
{
    datagram = o->ice_datagram();
}
catch(const Ice::NoEndpointException&)
{
    cerr << "No endpoint for datagram invocations" << endl;
}

// Down-cast to actual type.
//
PersonPrx datagramPerson = PersonPrx::uncheckedCast(datagram);

// Invoke an operation as a datagram.
//
try
{
    datagramPerson->someOp();
}
catch(const Ice::TwowayOnlyException&)
{
    cerr << "someOp() is not oneway" << endl;
}
```

As for the oneway example, you can alternatively choose to first do a safe down-cast to the actual type of interface and then obtain the datagram proxy, rather than relying on an unsafe down-cast, as shown above. However, doing so may be disadvantageous for two reasons:

- Safe down-casts are sent via a stream-oriented transport. This means that using a safe down-cast will result in opening a connection for the sole purpose of verifying that the target object has the correct type. This is expensive if all the other traffic to the object is sent via datagrams.
- If the proxy does not offer a stream-oriented transport, the `checkedCast` fails with a `NoEndpointException`, so you can use this approach only for proxies that offer both a UDP endpoint and a TCP/IP and/or SSL endpoint.

Using UDP Multicast

The UDP transport included in Ice for C++, Java and .NET also supports IP multicast. Assuming it's enabled on your host, using IP multicast in your application can be as simple as changing the host in the UDP endpoint to an IPv4 or IPv6 address in the multicast range:

```
# Object Adapter endpoint:
Discover.Endpoints=udp -h 239.255.1.1 -p 10000

# Corresponding proxy endpoint:
Discover.Proxy=discover:udp -h 239.255.1.1 -p 10000
```

You can optionally select the network interface to use for multicast endpoints by including the `--interface` option.

In one respect, using multicast in Ice is no different than using regular datagram invocations; all of the design considerations mentioned above still apply. However, the fact that there could be any number of listeners (or none at all) adds new possibilities for your application design. The Ice distribution includes a simple example of a multicast application in `demo/Ice/multicast`.

> Consider using the IceDiscovery plug-in if your objective in using multicast is the discovery of available servers.

See Also

- Terminology
- Oneway Invocations
- The Ice Protocol

Batched Invocations

Oneway and datagram invocations are normally sent as individual messages, that is, the Ice run time sends the oneway or datagram invocation to the server immediately, as soon as the client makes the call. If a client sends a number of oneway or datagram invocations in succession, the client-side run time traps into the OS kernel for each message, which is expensive. In addition, each message is sent with its own message header, that is, for *N* messages, the bandwidth for *N* message headers is consumed. In situations where a client sends a number of oneway or datagram invocations, the additional overhead can be considerable.

To avoid the overhead of sending many small messages, you can send oneway and datagram invocations in a batch: instead of being sent as a separate message, a batch invocation is placed into a client-side buffer by the Ice run time. Successive batch invocations are added to the buffer and accumulated on the client side until they are flushed, either explicitly by the client or automatically by the Ice run time.

On this page:

Proxy Methods for Batched Invocations

Several proxy methods support the use of batched invocations. In Slice, these methods would look as follows:

Slice

```
Object* ice_batchOneway();
Object* ice_batchDatagram();
void ice_flushBatchRequests();
```

The `ice_batchOneway` and `ice_batchDatagram` methods create a new proxy configured for batch invocations. Once you obtain a batch proxy, messages sent via that proxy are buffered by the proxy instead of being sent immediately. Once the client has invoked one or more operations on a batch proxy, it can call `ice_flushBatchRequests` to explicitly flush the batched invocations. This causes the batched messages to be sent "in bulk", preceded by a single message header. On the server side, batched messages are dispatched by a single thread, in the order in which they were written into the batch. This means that messages from a single batch cannot appear to be reordered in the server. Moreover, either all messages in a batch are delivered or none of them. (This is true even for batched datagrams.)

Asynchronous versions of `ice_flushBatchRequests` are also available; see the relevant language mapping for more information.

Batched invocations are queued by the proxy on which the request was invoked (this is true for all proxies except fixed proxies). It's important to be aware of this behavior for several reasons:

- Batched invocations queued on a proxy will be lost if that proxy is deallocated prior to being flushed
- Proxy instances maintain separate queues even if they refer to the same target object
- Using proxy factory methods may (or may not) create new proxy instances, which affects how batched invocations are queued. Consider this example:

C++11

```
communicator = ...;
auto batch = communicator->stringToProxy("...")->ice_batchOneway();
// Creates a batch oneway proxy
auto secureBatch = batch->ice_secure(); // Might create a new proxy
instance
batch->ice_ping();
secureBatch->ice_ping();
batch->ice_flushBatchRequests(); // Might also flush requests on
secureBatch
```

At run time, the `batch` and `secureBatch` variables might refer to the same proxy instance or two separate proxy instances, depending on the original stringified proxy's configuration. As a result, flushing requests using one variable may or may not flush the requests of the other.

Calling `ice_flushBatchRequests` on a proxy behaves like a oneway invocation in that automatic retries take place and an exception is raised if an error occurs while establishing a connection or sending the batch message.

Automatically Flushing Batched Invocations

The default behavior of the Ice run time, as governed by the configuration property `Ice.BatchAutoFlushSize`, automatically flushes batched invocations as soon as a batched request causes the accumulated message to exceed the specified limit. When this occurs, the Ice run time immediately flushes the existing batch of requests and begins a new batch with this latest request as its first element.

For batched oneway invocations, the value of `Ice.BatchAutoFlushSize` specifies the maximum message size in kilobytes; the default value is 1MB. In the case of batched datagram invocations, the maximum message size is the smaller of the system's maximum size for datagram packets and the value of `Ice.BatchAutoFlushSize`.

> The receiver's setting for `Ice.MessageSizeMax` determines the maximum size that the Ice run time will accept for an incoming protocol message. The sender's setting for `Ice.BatchAutoFlushSize` must not exceed this limit, otherwise the receiver will silently discard the entire batch.

Automatic flushing is enabled by default as a convenience for clients to ensure a batch never exceeds the configured limit. A client can track batch request activity, and even implement its own auto-flush logic, by installing an interceptor.

Batched Invocations for Fixed Proxies

A *fixed proxy* is a special form of proxy that an application explicitly creates for use with a specific bidirectional connection. Batched invocations on a fixed proxy are not queued by the proxy, as is the case for regular proxies, but rather by the connection associated with the fixed proxy. Automatic flushing continues to work as usual for batched invocations on fixed proxies, and you have three options for manually flushing:

- Calling `ice_flushBatchRequests` on a fixed proxy flushes all batched requests queued by its connection; this includes batched requests from other fixed proxies that share the same connection
- Calling `Connection::flushBatchRequests` flushes all batched requests queued by the target connection
- Calling `Communicator::flushBatchRequests` flushes all batched requests on all connections associated with the target communicator

> `Connection::flushBatchRequests` and `Communicator::flushBatchRequests` have no effect on batched requests queued by regular (non-fixed) proxies.

The synchronous versions of `flushBatchRequests` block the calling thread until the batched requests have been successfully written to the local transport. To avoid the risk of blocking, you must use the asynchronous versions instead (assuming they are supported by your

chosen language mapping).

Note the following limitations in case a connection error occurs:

- Any requests queued by that connection are lost
- Automatic retries are not attempted
- The proxy method `ice_flushBatchRequests` and `Connection::flushBatchRequests` will raise an exception, but `Communicator::flushBatchRequests` ignores any errors

The `Connection::flushBatchRequests` and `Communicator::flushBatchRequests` methods take an `Ice::CompressBatch` argument to specify under which conditions the batch should be compressed or not. The `Ice::CompressBatch` enumeration is shown below:

slice

```
["cpp:scoped", "objc:scoped"]
local enum CompressBatch
{
    Yes,
    No,
    BasedOnProxy
}
```

You can choose to always or never compress the batch with `Yes` or `No`. The `BasedOnProxy` value specifies to compress based on the proxies used to add requests to the batch. If at least one request was queued with a compressed fixed proxy (a proxy created with `ice_compress(true)` or if `Ice.Override.Compress` is enabled), the batch will be compressed.

Considerations for Batched Datagrams

For batched datagram invocations, you need to keep in mind that, if the data for the invocations in a batch substantially exceeds the PDU size of the network, it becomes increasingly likely for an individual UDP packet to get lost due to fragmentation. In turn, loss of even a single packet causes the entire batch to be lost. For this reason, batched datagram invocations are most suitable for simple interfaces with a number of operations that each set an attribute of the target object (or interfaces with similar semantics). Batched oneway invocations do not suffer from this risk because they are sent over stream-oriented transports, so individual packets cannot be lost.

If automatic flushing is enabled, Ice's default behavior uses the smaller of `Ice.BatchAutoFlushSize` and `Ice.UDP.SndSize` to determine the maximum size for a batch datagram message.

Compressing Batched Invocations

Batched invocations are more efficient if you also enable compression for the transport: many isolated and small messages are unlikely to compress well, whereas batched messages are likely to provide better compression because the compression algorithm has more data to work with.

> Regardless of whether you used batched messages or not, you should enable compression only on lower-speed links. For high-speed LAN connections, the CPU time spent doing the compression and decompression is typically longer than the time it takes to just transmit the uncompressed data.

Active Connection Management and Batched Invocations

As for oneway invocations, server-side Active Connection Management (ACM) can interfere with batched invocations over TCP or TCP-based transports (SSL, WebSocket, etc.). With server-side ACM enabled, it's possible for a server to close the connection at the wrong moment and not process a batch – with no indication being returned to the client that the batch was lost. We recommend that you either disable ACM for the server side, or enable ACM heartbeats in the client to ensure the connection remains active.

Batched Invocation Interceptors

Batch invocation interceptors allow you to implement your own auto-flush algorithm or receive notification when an auto-flush fails.

C++11 C++98 C# Java Java Compat Obj-C Python

```
namespace Ice
{
    class BatchRequest
    {
    public:
        virtual void enqueue() const = 0;
        virtual int getSize() const = 0;
        virtual const std::string& getOperation() const = 0;
        virtual const std::shared_ptr<Ice::ObjectPrx>& getProxy() const
= 0;
    };

    struct InitializationData
    {
        ...
        std::function<void(const Ice::BatchRequest& req, int count, int
size)> batchRequestInterceptor;
    };
}
```

```
namespace Ice
{
    class BatchRequest
    {
    public:
        virtual void enqueue() const = 0;
        virtual int getSize() const = 0;
        virtual const std::string& getOperation() const = 0;
        virtual const Ice::ObjectPrx& getProxy() const = 0;
    };

    class BatchRequestInterceptor : public IceUtil::Shared
    {
    public:
        virtual void enqueue(const BatchRequest&, int, int) = 0;
    };

    struct InitializationData
    {
        ...
        BatchRequestInterceptorPtr batchRequestInterceptor;
    };
}
```

```
namespace Ice
{
    public interface BatchRequest
    {
        void enqueue();
        int getSize();
        string getOperation();
        ObjectPrx getProxy();
    }

    public final class InitializationData
    {
    ...
        public System.Action<BatchRequest, int, int>
batchRequestInterceptor;
    }
}
```

```
package com.zeroc.Ice;

public interface BatchRequest
{
    void enqueue();
    int getSize();
    String getOperation();
    ObjectPrx getProxy();
}

@FunctionalInterface
public interface BatchRequestInterceptor
{
    void enqueue(BatchRequest request, int queueBatchRequestCount, int
queueBatchRequestSize);
}

public final class InitializationData
{
    ...
    public BatchRequestInterceptor batchRequestInterceptor;
}
```

```
package Ice;

public interface BatchRequest
{
    void enqueue();
    int getSize();
    String getOperation();
    ObjectPrx getProxy();
}
public interface BatchRequestInterceptor
{
    void enqueue(BatchRequest request, int queueBatchRequestCount, int
queueBatchRequestSize);
}

public final class InitializationData
{
    ...
    public BatchRequestInterceptor batchRequestInterceptor;
}
```

```
@protocol ICEBatchRequest <NSObject>
-(void) enqueue;
-(int) getSize;
-(NSString*) getOperation;
-(id<ICEObjectPrx>) getProxy;
@end

@interface ICEInitializationData : NSObject
...
@property(copy, nonatomic)
void(^batchRequestInterceptor)(id<ICEBatchRequest>, int, int);
@end
```

```
class BatchRequest(object):
    def getSize():
        ...
    def getOperation():
        ...

    def getProxy():
        ...

    def enqueue():
        ...

initData = Ice.InitializationData()
initData.batchRequestInterceptor = lambda req, count, size: ...
```

You install an interceptor by setting the `batchRequestInterceptor` member of the `InitializationData` object that the application constructs when initializing a new communicator. The Ice run time invokes the interceptor for each batch request, passing the following arguments:

- `req` - An object representing the batch request being queued
- `count` - The number of requests currently in the queue
- `size` - The number of bytes consumed by the requests currently in the queue

The request represented by `req` is not included in the `count` and `size` figures.

A batch request is not queued until the interceptor calls `BatchRequest::enqueue`. The minimal interceptor implementation is therefore:

C++11

```
initData.batchRequestInterceptor = [](const Ice::BatchRequest& req, int
count, int size)
{
    req.enqueue();
};
```

A more sophisticated implementation might use its own logic for automatically flushing queued requests:

C++11

```
int limit =
initData.properties->getPropertyAsInt("Ice.BatchAutoFlushSize");
initData.batchRequestInterceptor = [limit](const Ice::BatchRequest& req,
int count, int size)
{
    if(size + req.getSize() > limit)
    {
        req.getProxy()->ice_flushBatchRequestsAsync([=](const
Ice::Exception& ex) { /* Handle error */ });
    }
    req.enqueue();
};
```

In this example, the implementation consults the existing Ice property `Ice.BatchAutoFlushSize` to determine the limit that triggers an automatic flush. If a flush is necessary, the interceptor can obtain the relevant proxy by calling `getProxy` on the `BatchRequest` object.

Specifying your own exception handler when calling `ice_flushBatchRequestsAsync` gives you the ability to take action if a failure occurs (Ice's default automatic flushing implementation ignores any errors). Aside from logging a message, your options are somewhat limited because it's not possible for the interceptor to force a retry.

> For datagram proxies, we strongly recommend using a maximum queue size that is smaller than the network MTU to minimize the risk that datagram fragmentation could cause an entire batch to be lost.

See Also

- Oneway Invocations
- Datagram Invocations
- Communicator
- Using Connections
- The Ice Protocol
- Protocol Compression
- Active Connection Management

Server-Side Features

This section presents all APIs and features that are specific to server applications.

Client-specific features are presented in Client-Side Features, while transverse features best understood while considering both sides of a client-server interaction are described in Client-Server Features.

Topics

Object Adapters

A communicator contains one or more object adapters. An object adapter sits at the boundary between the Ice run time and the server application code and has a number of responsibilities:

- Maps Ice objects to servants for incoming requests and dispatches the requests to the application code in each servant (that is, an object adapter implements an up-call interface that connects the Ice run time and the application code in the server).
- Assists in life cycle operations so Ice objects and servants can be created and existing destroyed without race conditions.
- Provides one or more transport endpoints. Clients access the Ice objects provided by the adapter via those endpoints. (It is also possible to create an object adapter without endpoints. In this case the adapter is used for bidirectional callbacks.

Each object adapter has one or more servants that incarnate Ice objects, as well as one or more transport endpoints. If an object adapter has more than one endpoint, all servants registered with that adapter respond to incoming requests on any of the endpoints. In other words, if an object adapter has multiple transport endpoints, those endpoints represent alternative communication paths to the same set of objects (for example, via different transports).

Each object adapter belongs to exactly one communicator (but a single communicator can have many object adapters). Each object adapter has a name that distinguishes it from all other object adapters in the same communicator.

Each object adapter can optionally have its own thread pool, enabled via the `<adapter-name>.ThreadPool.Size` property. If so, client invocations for that adapter are dispatched in a thread taken from the adapter's thread pool instead of using a thread from the communicator's server thread pool.

Servants are the physical manifestation of an Ice object, that is, they are entities that are implemented in a concrete programming language and instantiated in the server's address space. Servants provide the server-side behavior for operation invocations sent by clients. The same servant can be registered with one or more object adapters.

Topics

See Also

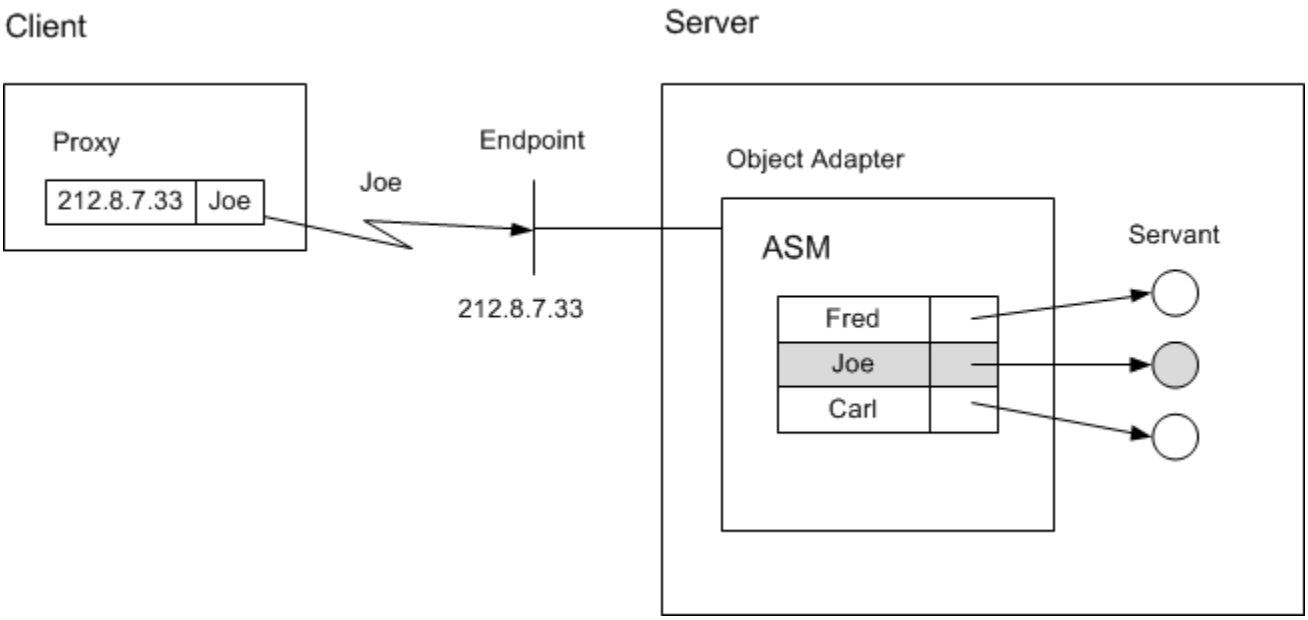
The Active Servant Map

Each object adapter maintains a data structure known as the active servant map.

On this page:

Role of the Active Servant Map

The *active servant map* (or *ASM*, for short) is a lookup table that maps object identities to servants: for C++, the lookup value is a smart pointer to the corresponding servant's location in memory; for Java and C#, the lookup value is a reference to the servant. When a client sends an operation invocation to the server, the request is targeted at a specific transport endpoint. Implicitly, the transport endpoint identifies the object adapter that is the target of the request (because no two object adapters can be bound to the same endpoint). The proxy via which the client sends its request contains the object identity for the corresponding object, and the client-side run time sends this object identity over the wire with the invocation. In turn, the object adapter uses that object identity to look in its ASM for the correct servant to dispatch the call to, as shown below:



Binding a request to the correct servant.

The process of associating a request via a proxy to the correct servant is known as *binding*. The scenario depicted in the illustration shows direct binding, in which the transport endpoint is embedded in the proxy. Ice also supports an indirect binding mode, in which the correct transport endpoints are provided by a location service.

If a client request contains an object identity for which there is no entry in the adapter's ASM, the adapter returns an `ObjectNotExistException` to the client (unless you use a default servant or servant locator).

Design Considerations for the Active Servant Map

Using an adapter's ASM to map Ice objects to servants has a number of design implications:

- Each Ice object is represented by a different servant.

  > It is possible to register a single servant with multiple identities. However, there is little point in doing so because a default servant achieves the same thing.

- All servants for all Ice objects are permanently in memory.

Using a separate servant for each Ice object in this fashion is common to many server implementations: the technique is simple to implement and provides a natural mapping from Ice objects to servants. Typically, on start-up, the server instantiates a separate servant for each Ice object, activates each servant, and then calls `activate` on the object adapter to start the flow of requests.

There is nothing wrong with the above design, provided that two criteria are met:

1. The server has sufficient memory available to keep a separate servant instantiated for each Ice object at all times.
2. The time required to initialize all the servants on start-up is acceptable.

For many servers, neither criterion presents a problem: provided that the number of servants is small enough and that the servants can be initialized quickly, this is a perfectly acceptable design. However, the design does not scale well: the memory requirements of the server grow linearly with the number of Ice objects so, if the number of objects gets too large (or if each servant stores too much state), the server runs out of memory.

Ice offers two APIs that help you scale servers to larger numbers of objects: *servant locators* and *default servants*. A default servant is essentially a simplified version of a servant locator that satisfies the majority of use cases, whereas a servant locator provides more flexibility for those applications that require it.

See Also

- Servant Locators
- Default Servants
- Locators

Creating an Object Adapter

`ObjectAdapter` Local Interface

An object adapter is an instance of the local interface `ObjectAdapter`.

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        string getName();
        Communicator getCommunicator();

        // ...
    }
}
```

The `ObjectAdapter` operations behave as follows:

- The `getName` operation returns the name of the adapter as passed to one of the communicator operations `createObjectAdapter`, `createObjectAdapterWithEndpoints`, or `createObjectAdapterWithRouter`.
- The `getCommunicator` operation returns the communicator that was used to create the adapter.

Note that there are other operations in the `ObjectAdapter` interface; we will explore these throughout the remainder of this discussion.

Object Adapter Factory Operations

You create an object adapter by calling one of several operations on `Communicator`:

```
module Ice
{
    local interface Communicator
    {
        ObjectAdapter createObjectAdapter(string name);
        ObjectAdapter createObjectAdapterWithEndpoints(string name,
string endpoints);
        ObjectAdapter createObjectAdapterWithRouter(string name, Router*
rtr);

        // ...
    }
}
```

These `Communicator` operations behave as follows:

- `createObjectAdapter` creates a new object adapter associated with this communicator. Each object adapter is associated with

zero or more transport endpoints. Typically, an object adapter has a single transport endpoint.

> An object adapter can also offer multiple endpoints. If so, these endpoints each lead to the same set of objects and represent alternative means of accessing these objects. This is useful, for example, if a server is behind a firewall but must offer access to its objects to both internal and external clients; by binding the adapter to both the internal and external interfaces, the objects implemented in the server can be accessed via either interface.
>
> An object adapter can also have no endpoint at all. In that case, the adapter can only be reached via collocated invocations originating from the same communicator as is used by the adapter.

An application normally needs to configure an object adapter with endpoints or a router. Calling `createObjectAdapter` with a non-empty value for `name` means the new object adapter will check the communicator's configuration for properties, using its name as prefix, including:

- *name*`.Endpoints` - defines one or more object adapter endpoints
- *name*`.Router` - specifies the stringified proxy of a router

`createObjectAdapter` raises an exception if neither is defined and name is not empty.

Passing an empty value for `name` to `createObjectAdapter` creates a nameless object adapter that can only be used for collocated invocations or callbacks over bidirectional connections.

- `createObjectAdapterWithEndpoints` allows you to specify the object adapter's endpoints directly, overriding any setting for *name*`.Endpoints`. Typically, you should use `createObjectAdapter` in preference to `createObjectAdapterWithEndpoints`. Doing so keeps transport-specific information, such as host names and port numbers, out of the source code and allows you to reconfigure the application by changing a property (and so avoid recompilation when a transport endpoint needs to be changed).
- `createObjectAdapterWithRouter` accepts a router proxy and overrides *name*`.Router`. `createObjectAdapterWithRouter` creates a routed object adapter that allows clients to receive callbacks from servers that are behind a router.

See Also

- Communicator

Servant Activation and Deactivation

The term *servant activation* refers to making the presence of a servant for a particular Ice object known to the Ice run time. Activating a servant adds an entry to the Active Servant Map (ASM). Another way of looking at servant activation is to think of it as creating a link between the identity of an Ice object and the corresponding programming-language servant that handles requests for that Ice object. Once the Ice run time has knowledge of this link, it can dispatch incoming requests to the correct servant. Without this link, that is, without a corresponding entry in the ASM, an incoming request for the identity results in an `ObjectNotExistException`. While a servant is activated, it is said to *incarnate* the corresponding Ice object.

The inverse operation is known as *servant deactivation*. Deactivating a servant removes an entry for a particular identity from the ASM. Thereafter, incoming requests for that identity are no longer dispatched to the servant and result in an `ObjectNotExistException`.

The object adapter offers a number of operations for managing servant activation and deactivation:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        // ...

        Object* add(Object servant, Identity id);
        Object* addWithUUID(Object servant);
        Object  remove(Identity id);
        Object  find(Identity id);
        Object  findByProxy(Object* proxy);

        // ...
    }
}
```

The operations behave as follows:

- `add`

  The `add` operation adds a servant with the given identity to the ASM. Requests are dispatched to that servant as soon as `add` is called. The return value is the proxy for the Ice object incarnated by that servant. The proxy embeds the identity passed to `add`.

  You cannot call `add` with the same identity more than once: attempts to add an already existing identity to the ASM result in an `AlreadyRegisteredException`. (It does not make sense to add two servants with the same identity because that would make it ambiguous as to which servant should handle incoming requests for that identity.)

  Note that it is possible to activate the same servant multiple times with different identities. In that case, the same single servant incarnates multiple Ice objects. We explore the ramifications of this in more detail in our discussion of server implementation techniques.

- `addWithUUID`

  The `addWithUUID` operation behaves the same way as the `add` operation but does not require you to supply an identity for the servant. Instead, `addWithUUID` generates a UUID as the identity for the corresponding Ice object. You can retrieve the generated identity by calling the `ice_getIdentity` operation on the returned proxy. `addWithUUID` is useful to create identities for temporary objects, such as short-lived session objects. (You can also use `addWithUUID` for persistent objects that do not have a natural identity, as we have done for the file system application.)

- `remove`

  The `remove` operation breaks the association between an identity and its servant by removing the corresponding entry from the ASM; it returns a reference to the removed servant.

Once the servant is deactivated, new incoming requests for the removed identity cause the client to receive an `ObjectNotExistException`. Requests that are executing inside the servant at the time `remove` is called are allowed to complete normally. Once the last request for the servant is complete, the object adapter drops its reference (or smart pointer, for C++) to the servant. At that point, the servant becomes available for garbage collection (or is destroyed, for C++), provided there are no other references or smart pointers to the servant. The net effect is that a deactivated servant is destroyed once it becomes idle.

Deactivating an object adapter implicitly calls `remove` on its active servants.

- `find`

  The `find` operation performs a lookup in the ASM and returns the servant for the specified object identity. If no servant with that identity is registered, the operation returns null. Note that `find` does not consult any servant locators or default servants.

- `findByProxy`

  The `findByProxy` operation performs a lookup in the ASM and returns the servant with the object identity and facet that are embedded in the proxy. If no such servant is registered, the operation returns null. Note that `findByProxy` does not consult any servant locators or default servants.

See Also

- The Active Servant Map
- Object Identity
- Object Adapter States
- Server Implementation Techniques
- Servant Locators
- Default Servants

Object Adapter States

On this page:

Object Adapter State Transitions

An object adapter has a number of processing states:

- **Holding**
  In this state, any incoming requests for the adapter are held, that is, not dispatched to servants.

  For TCP/IP (and other stream-oriented protocols), the server-side run time stops reading from the corresponding transport endpoint while the adapter is in the holding state. In addition, it also does not accept incoming connection requests from clients. This means that if a client sends a request to an adapter that is in the holding state, the client eventually receives a `TimeoutException` or `ConnectTimeoutException` (unless the adapter is placed into the active state before the timer expires).

  For UDP, client requests that arrive at an adapter that is in the holding state are thrown away.

  Immediately after creation of an adapter, the adapter is in the holding state. This means that requests are not dispatched until you place the adapter into the active state.

- **Active**
  In this state, the adapter accepts incoming requests and dispatches them to servants. A newly-created adapter is initially in the holding state. The adapter begins dispatching requests as soon as you place it into the active state.

  You can transition between the active and the holding state as many times as you wish.

- **Inactive**
  In this state, the adapter has conceptually been destroyed (or is in the process of being destroyed). Deactivating an adapter destroys all transport endpoints that are associated with the adapter. Requests that are executing at the time the adapter is placed into the inactive state are allowed to complete, but no new requests are accepted. (New requests are rejected with an exception). Any attempt to use a deactivated object adapter results in an `ObjectAdapterDeactivatedException`.

> The distinction `Holding` vs Active `applies` only to requests dispatched through endpoints configured on the object adapter. It does not apply to requests received from bidirectional connections, nor does it apply to collocated dispatches (unless you disable collocation optimization).
>
> An object adapter configured with a router accepts only requests received over the (bidirectional) connection to the router, and collocated dispatches.

Changing Object Adapter States

The `ObjectAdapter` interface offers operations that allow you to change the adapter state, as well as to wait for a state change to be complete:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        // ...

        void activate();
        void hold();
        void waitForHold();
        void deactivate();
        void waitForDeactivate();
        void isDeactivated();
        void destroy();

        // ...
    }
}
```

The operations behave as follows:

- `activate`
  The `activate` operation places the adapter into the active state. Activating an adapter that is already active has no effect. The Ice run time starts dispatching requests to servants for the adapter as soon as `activate` is called.

- `hold`
  The `hold` operation places the adapter into the holding state. Requests that arrive after calling `hold` are held as described above. Requests that are in progress at the time `hold` is called are allowed to complete normally. Note that `hold` returns immediately without waiting for currently executing requests to complete.

- `waitForHold`
  The `waitForHold` operation suspends the calling thread until the adapter has completed its transition to the holding state, that is, until all currently executing requests have finished. You can call `waitForHold` from multiple threads, and you can call `waitForHold` while the adapter is in the active state. If you call `waitForHold` on an adapter that is already in the holding state, `waitForHold` returns immediately.

- `deactivate`
  The `deactivate` operation initiates deactivation of the adapter: requests that arrive after calling `deactivate` are rejected, but currently executing requests are allowed to complete. Once all requests have completed, the adapter enters the inactive state. Note that `deactivate` returns immediately without waiting for the currently executing requests to complete. A deactivated adapter cannot be reactivated; you can create a new adapter with the same name, but only after calling `destroy` on the existing adapter. Any attempt to use a deactivated object adapter results in an `ObjectAdapterDeactivatedException`. During deactivation, the association with bidirectional connections (if any) is cleared. Deactivation also makes the adapter ineligible for collocated dispatches.

- `waitForDeactivate`
  The `waitForDeactivate` operation suspends the calling thread until the adapter has completed its transition to the inactive state, that is, until all currently executing requests have completed. You can call `waitForDeactivate` from multiple threads, and you can call `waitForDeactivate` while the adapter is in the active or holding state. Calling `waitForDeactivate` on an adapter that is in the inactive state does nothing and returns immediately.

- `isDeactivated`
  The `isDeactivated` operation returns true if `deactivate` has been invoked on the adapter. A return value of true does not necessarily indicate that the adapter has fully transitioned to the inactive state, only that it has begun this transition. Applications that need to know when deactivation is completed can use `waitForDeactivate`.

- `destroy`
  The `destroy` operation deactivates the adapter and releases all of its resources. Internally, `destroy` invokes `deactivate` followe

d by `waitForDeactivate`, therefore the operation blocks until all currently executing requests have completed. Furthermore, any servants associated with the adapter are destroyed, all transport endpoints are closed, and the adapter's name becomes available for reuse.
Destroying a communicator implicitly destroys all of its object adapters. Invoking `destroy` on an adapter is only necessary when you need to ensure that its resources are released prior to the destruction of its communicator.

Placing an adapter into the holding state is useful, for example, if you need to make state changes in the server that require the server (or a group of servants) to be idle. For example, you could place the implementation of your servants into a dynamic library and upgrade the implementation by loading a newer version of the library at run time without having to shut down the server.

Similarly, waiting for an adapter to complete its transition to the inactive state is useful if your server needs to perform some final clean-up work that cannot be carried out until all executing requests have completed.

Note that you can create an object adapter with the same name as a previous object adapter, but only once `destroy` on the previous adapter has completed.

See Also

- Collocated Invocation and Dispatch
- Bidirectional Connections

Object Adapter Endpoints

An object adapter maintains two sets of transport endpoints. One set identifies the network interfaces on which the adapter listens for new connections, and the other set is embedded in proxies created by the adapter and used by clients to communicate with it. We will refer to these sets of endpoints as the *physical endpoints* and the *published endpoints*, respectively. In most cases these sets are identical, but there are situations when they must be configured independently.

On this page:

Physical Object Adapter Endpoints

An object adapter's physical endpoints identify the network interfaces on which it receives requests from clients. These endpoints are configured via the *name*`.Endpoints` property, or they can be specified explicitly when creating an adapter using the operation `createObjectAdapterWithEndpoints`. The endpoint syntax generally consists of a transport protocol followed by an optional host name and port.

If a host name is specified, the object adapter listens only on the network interface associated with that host name. If no host name is specified but the property `Ice.Default.Host` is defined, the object adapter uses the property's value as the host name. Finally, if a host name is not specified, and the property `Ice.Default.Host` is undefined, the object adapter listens on all available network interfaces, including the loopback interface. You may also force the object adapter to listen on all interfaces by using one of the host names `0.0.0.0` or `*`. The adapter does *not* expand the list of interfaces when it is initialized. Instead, if no host is specified, or you use `-h *` or `-h 0.0.0.0`, the adapter binds to `INADDR_ANY` to listen for incoming requests.

If the host name refers to a DNS name which is configured with multiple addresses, the object adapter will listen on the network interfaces identified by each address. All the addresses should refer to local network interfaces or the object adapter creation will fail.

If you want an adapter to accept requests on certain network interfaces, you must specify a separate endpoint for each interface. For example, the following property configures a single endpoint for the adapter named `MyAdapter`:

```
MyAdapter.Endpoints=tcp -h 10.0.1.1 -p 9999
```

This endpoint causes the adapter to accept requests on the network interface associated with the IP address `10.0.1.1` at port `9999`. Note however that this adapter configuration does not accept requests on the loopback interface (the one associated with address `127.0.0.1`). If both addresses must be supported, then both must be specified explicitly, as shown below:

```
MyAdapter.Endpoints=tcp -h 10.0.1.1 -p 9999:tcp -h 127.0.0.1 -p 9999
```

If these are the only two network interfaces available on the host, then a simpler configuration omits the host name altogether, causing the object adapter to listen on both interfaces automatically:

```
MyAdapter.Endpoints=tcp -p 9999
```

If you want to make your configuration more explicit, you can use one of the special host names mentioned earlier:

```
MyAdapter.Endpoints=tcp -h * -p 9999
```

One advantage of this last example is that it ensures the object adapter *always* listens on all interfaces, even if a definition for `Ice.Default.Host` is later added to your configuration. Without an explicit host name, the addition of `Ice.Default.Host` could potentially change the interfaces on which the adapter is listening. For diagnostic purposes, you can determine the set of local addresses that Ice substitutes for the

wildcard address by setting the property `Ice.Trace.Network=3` and reviewing the server's log output.

Careful consideration must also be given to the selection of a port for an endpoint. If no port is specified, the adapter uses a port that is selected (essentially at random) by the operating system, meaning the adapter will likely be using a different port each time the server is restarted. Whether that behavior is desirable depends on the application, but in many applications a client has a proxy containing the adapter's endpoint and expects that proxy to remain valid indefinitely. Therefore, an endpoint generally should contain a fixed port to ensure that the adapter is always listening at the same port.

However, there are certain situations where a fixed port is not required. For example, an adapter whose servants are transient does not need a fixed port, because the proxies for those objects are not expected to remain valid past the lifetime of the server process. Similarly, a server using indirect binding via IceGrid does not need a fixed port because its port is never published.

Published Object Adapter Endpoints

An object adapter publishes its endpoints in the proxies it creates, but it is not always appropriate to publish the adapter's physical endpoints in a proxy. For example, suppose a server is running on a host in a private network, protected from the public network by a firewall that can forward network traffic to the server. The adapter's physical endpoints must use the private network's address scheme, but a client in the public network would be unable to use those endpoints if they were published in a proxy. In this scenario, the adapter must publish endpoints in its proxies that direct the client to the firewall instead.

The published endpoints are configured using the adapter property *name*`.PublishedEndpoints`.

If this property is not defined, the adapter publishes its physical endpoints by default, with two exceptions:

- endpoints for the loopback address (`127.0.0.1`) are not published unless the loopback interface is the only interface, or `127.0.0.1` (or `loopback`) is explicitly listed as an endpoint with the `-h` option. Otherwise, to force the inclusion of loopback endpoints when they would normally be excluded, you must define *name*`.PublishedEndpoints` explicitly,
- endpoints with DNS names and no fixed port number are expanded to multiple endpoints if the DNS name refers to multiple addresses. There will be one published endpoint with the system allocated port per address. For example, `tcp -h corphost` coul d be expanded to `tcp -h 192.168.1.64 -p 64567:tcp -h 192.168.2.67 -p 64568`.

As an example, the properties below configure the adapter named `MyAdapter` with physical and published endpoints:

```
MyAdapter.Endpoints=tcp -h 10.0.1.1 -p 9999
MyAdapter.PublishedEndpoints=tcp -h corpfw -p 25000
```

This example assumes that clients connecting to host `corpfw` at port `25000` are forwarded to the adapter's endpoint in the private network.

Another use case of published endpoints is for replicated servers. Suppose we have two instances of a stateless server running on separate hosts in order to distribute the load between them. We can supply the client with a bootstrap proxy containing the endpoints of both servers, and the Ice run time in the client will select one of the servers at random when a connection is established. However, should the client invoke an operation on a server that returns a proxy for another object, that proxy would normally contain only the endpoint of the server that created it. Invocations on the new proxy would always be directed at the same server, reducing the opportunity for load balancing.

We can alleviate this situation by configuring the adapters to publish the endpoints of both servers. For example, here is a configuration for the server on host `Sun1`:

```
MyAdapter.Endpoints=tcp -h Sun1 -p 9999
MyAdapter.PublishedEndpoints=tcp -h Sun1 -p 9999:tcp -h Sun2 -p 9999
```

Similarly, the configuration for host `Sun2` retains the same published endpoints:

```
MyAdapter.Endpoints=tcp -h Sun2 -p 9999
MyAdapter.PublishedEndpoints=tcp -h Sun1 -p 9999:tcp -h Sun2 -p 9999
```

For troubleshooting purposes, you can examine the published endpoints for an object adapter by setting the property `Ice.Trace.Network=3`. Note however that this setting generates significant trace information about the Ice run time's network activity, therefore you may not

want to use this setting by default.

You can also change the published endpoints at run time, by calling `setPublishedEndpoints` on `ObjectAdapter`:

Slice

```
local interface ObjectAdapter
{
    void setPublishedEndpoints(EndpointSeq newEndpoints);
    // ...
}
```

Refreshing Object Adapter Endpoints

A host's network interfaces may change over time, for example, if a laptop moves in and out of range of a wireless network. The object adapter provides an operation to refresh its list of interfaces:

Slice

```
local interface ObjectAdapter
{
    void refreshPublishedEndpoints();
    // ...
}
```

Calling `refreshPublishedEndpoints` causes the object adapter to update its internal list of available network interfaces and to use this updated information in the *`name`*`.PublishedEndpoints` property. This allows you to react to changing network interfaces while an object adapter is in use, but your application code is responsible for determining when it is necessary to call this operation.

Note that `refreshPublishedEndpoints` takes effect only for object adapters that specify published endpoints without a host, or that set the published endpoints to `-h *` or `-h 0.0.0.0`.

Timeouts in Object Adapter Endpoints

As a defense against hostile clients, we recommend that you specify a timeout for your physical object adapter endpoints. The timeout value you select affects tasks that the Ice run time normally does not expect to block for any significant amount of time, such as writing a reply message to a socket or waiting for SSL negotiation to complete. If you don't specify a timeout for an endpoint, the Ice run time uses the value of the `Ice.Default.Timeout` property. You can also specify the value `infinite` in an endpoint to wait indefinitely, but be aware that this could allow malicious or misbehaving clients to consume excessive resources such as file descriptors.

Specifying a timeout in an object adapter endpoint is done exactly as in a proxy endpoint, using the `-t` option:

```
MyAdapter.Endpoints=tcp -p 9999 -t 5000
```

In this example, we specify a timeout of five seconds.

Unless overridden by published endpoints, the timeout specified in your object adapter endpoint also appears in the endpoints of any proxies created by that object adapter. This feature allows you to provide a default timeout value for clients that use your object adapter, although the client's code or configuration can cause it to use a different timeout in practice.

Discovering Object Adapter Endpoints

The object adapter provides operations for retrieving the physical and published endpoints:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        // ...

        EndpointSeq getEndpoints();
        EndpointSeq getPublishedEndpoints();

        // ...
    }
}
```

The sequences that are returned contain `Endpoint` objects representing the adapter's physical and published endpoints, respectively.

A Router's Effect on Object Adapter Endpoints

If an object adapter is configured with a router, the adapter's published endpoints are those provided by the router's server proxy. Calling `setPublishedEndpoints` on such an object adapter will raise an illegal argument exception and `refreshPublishedEndpoints` will retrieve the latest router's server endpoints. See Routers for additional information.

See Also

- Proxy and Endpoint Syntax
- Object Adapter Properties
- Ice.Default.*
- IceGrid
- Using Connections
- Communicator
- Routers

Creating Proxies with an Object Adapter

Proxies are created as a side-effect of using the servant activation operations, but the life cycle of proxies is completely independent from that of servants. The `ObjectAdapter` interface provides several operations for creating a proxy for an object, regardless of whether a servant is currently activated for that object's identity:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        Object* createProxy(Identity id);
        Object* createDirectProxy(Identity id);
        Object* createIndirectProxy(Identity id);
        // ...
    }
}
```

These operations are described below:

- `createProxy`
  The `createProxy` operation returns a new proxy for the object with the given identity. The adapter's configuration determines whether the return value is a direct proxy or an indirect proxy. If the adapter is configured with an adapter ID, the operation returns an indirect proxy that refers to the adapter ID. If the adapter is also configured with a replica group ID, the operation returns an indirect proxy that refers to the replica group ID. Otherwise, if an adapter ID is not defined, `createProxy` returns a direct proxy containing the adapter's published endpoints.
- `createDirectProxy`
  The `createDirectProxy` operation returns a direct proxy containing the adapter's published endpoints.
- `createIndirectProxy`
  The `createIndirectProxy` operation returns an indirect proxy. If the adapter is configured with an adapter ID, the returned proxy refers to that adapter ID. Otherwise, the proxy refers only to the object's identity.

In contrast to `createProxy`, `createIndirectProxy` does not use the replica group ID. Therefore, the returned proxy always refers to a specific replica.

After using one of the operations discussed above to create a proxy, you will receive a proxy that is configured by default for twoway invocations. If you require the proxy to have a different configuration, you can use the proxy factory methods to create a new proxy with the desired configuration. As an example, the code below demonstrates how to configure the proxy for oneway invocations:

C++11 C++98

```
std::shared_ptr<Ice::ObjectAdapter> adapter = ...;
Ice::Identity id = ...;
auto proxy = adapter->createProxy(id)->ice_oneway();
```

```
Ice::ObjectAdapterPtr adapter = ...;
Ice::Identity id = ...;
Ice::ObjectPrx proxy = adapter->createProxy(id)->ice_oneway();
```

You can also instruct the object adapter to use a different default proxy configuration by setting the property *`name`*`.ProxyOptions`. For example, the following property causes the object adapter to return proxies that are configured for oneway invocations by default:

```
MyAdapter.ProxyOptions=-o
```

See Also

- Servant Activation and Deactivation
- Object Life Cycle
- Object Identity
- Terminology
- Object Adapter Endpoints
- Object Adapter Replication
- Proxy Methods
- Object Adapter Properties

Using Multiple Object Adapters

A typical server rarely needs to use more than one object adapter. If you are considering using multiple object adapters, we suggest that you check whether any of the considerations in the list below apply to your situation:

- You need fine-grained control over which objects are accessible. For example, you could have an object adapter with only secure endpoints to restrict access to some administrative objects, and another object adapter with non-secure endpoints for other objects. Because an object adapter is associated with one or more transport endpoints, you can firewall a particular port, so objects associated with the corresponding endpoint cannot be reached unless the firewall rules are satisfied.
- You need control over the number of threads in the pools for different sets of objects in your application. For example, you may not need concurrency on the objects connected to a particular object adapter, and multiple object adapters, each with its own thread pool, can be useful to solve deadlocks.
- You want to be able to temporarily disable processing new requests for a set of objects. This can be accomplished by placing an object adapter in the holding state.
- You want to set up different request routing when using an Ice router with Glacier2.

If none of the preceding items apply, chances are that you do not need more than one object adapter.

See Also

- Object Adapter Thread Pools
- Nested Invocations
- Object Adapter States
- Glacier2

The Current Object

Up to now, we have tacitly ignored the trailing parameter of type `Ice::Current` that is passed to each skeleton operation on the server side. The `Current` structure is defined as follows:

Slice

```
module Ice
{
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Idempotent }

    local struct Current
    {
        ObjectAdapter   adapter;
        Connection      con;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
        Context         ctx;
        int             requestId;
        EncodingVersion encoding;
    }
}
```

Note that the `Current` value provides access to information about the currently executing request to the implementation of an operation in the server:

- `adapter`
  The `adapter` member provides access to the object adapter via which the request is being dispatched. In turn, the adapter provides access to its communicator (via the `getCommunicator` operation).
- `con`
  The `con` member provides information about the connection over which this request was received.
- `id`
  The `id` member provides the object identity for the current request.
- `facet`
  The `facet` member provides access to the facet for the request.
- `operation`
  The `operation` member contains the name of the operation that is being invoked. Note that the operation name may indicate one of the operations on `Ice::Object`, such as `ice_ping` or `ice_isA`. (`ice_isA` is invoked if a client performs a `checkedCast`.)
- `mode`
  The `mode` member contains the invocation mode for the operation (`Normal` or `Idempotent`), which influences the retry behavior of the Ice run time.
- `ctx`
  The `ctx` member contains the current request context for the invocation.
- `requestId`
  The Ice protocol uses request IDs to associate replies with their corresponding requests. The `requestId` member provides this ID. For oneway requests (which do not have replies), the request ID is `0`. For collocated requests (which do not use the Ice protocol), the request ID is `-1`.

- `encoding`
  The encoding version used to encode the input and output parameters.

If you implement your server such that it uses a separate servant for each Ice object, the contents of `Current` are not particularly interesting. (You would most likely access `Current` to read the `adapter` member, for example, to activate or deactivate a servant.) However, as we will see in our discussion of default servants and servant locators, the `Current` object is essential for more sophisticated (and more scalable) servant implementations.

See Also

- Object Identity
- Default Servants
- Servant Locators
- Request Contexts
- Versioning
- Using Connections
- The Ice Protocol

Servant Locators

In a nutshell, a servant locator is a local object that you implement and attach to an object adapter. Once an adapter has a servant locator, it consults its active servant map (ASM) to locate a servant for an incoming request as usual. If a servant for the request can be found in the ASM, the request is dispatched to that servant. However, if the ASM does not have an entry for the object identity of the request, the object adapter calls back into the servant locator to ask it to provide a servant for the request. The servant locator either:

- locates an existing servant or instantiates a new servant and passes it to the Ice run time, in which case the request is dispatched to that servant, or
- the servant locator indicates failure to locate a servant to the Ice run time, in which case the client receives an `ObjectNotExistException`.

This simple mechanism allows us to scale servers to provide access to an unlimited number of Ice objects. For example, instead of instantiating a separate servant for each and every Ice object in existence, the server can instantiate servants only for those Ice objects that are actually used by clients. Similar to a Default Servant, the server can also instantiate only one single servant that the locator returns for a group of Ice objects with similar properties.

Servant locators are most commonly used by servers that provide access to databases: typically, the number of entries in the database is far larger than what the server can hold in memory. Servant locators allow the server to only instantiate servants for those Ice objects that are actually used by clients.

Another common use for servant locators is in servers that are used for process control or network management: in that case, there is no database but, instead, there is a potentially very large number of devices or network elements that must be controlled via the server. Otherwise, this scenario is the same as for large databases: the number of Ice objects exceeds the number of servants that the server can hold in memory and, therefore, requires an approach that allows the number of instantiated servants to be less than the number of Ice objects.

Topics

See Also

- Object Adapters
- The Active Servant Map
- Object Identity
- Object Life Cycle
- Default Servants

The ServantLocator Interface

A servant locator has the following interface:

Slice

```
module Ice
{
    local interface ServantLocator
    {
        ["java:UserException"]
        Object locate(Current curr, out LocalObject cookie);

        ["java:UserException"]
        void finished(Current curr, Object servant, LocalObject cookie);

        void deactivate(string category);
    }
}
```

Note that `ServantLocator` is a local interface. To create an actual implementation of a servant locator, you must define a class that is derived from `Ice::ServantLocator` and provide implementations of the `locate`, `finished`, and `deactivate` operations. The Ice run time invokes the operations on your derived class as follows:

- `locate`

  Whenever a request arrives for which no entry exists in the active servant map (ASM), the Ice run time calls `locate` and supplies the `Current` object for the request. The implementation of `locate` (which you provide as part of the derived class) is supposed to return a servant that can process the incoming request. Your implementation of `locate` can behave in three possible ways:
  1. Locate an existing or instantiate a new servant, and return this servant for the current request. In this case, the Ice run time dispatches the request to this servant.
  2. Return null. In this case, the Ice run time raises an `ObjectNotExistException` in the client.
  3. Throw a run-time exception. In this case, the Ice run time propagates the thrown exception back to the client. Keep in mind that all run-time exceptions, apart from `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`, are presented as `UnknownLocalException` to the client.

  You can also throw user exceptions from `locate`. If the user exception is in the corresponding operation's exception specification, that user exception is returned to the client. User exceptions thrown by `locate` that are not listed in the exception specification of the corresponding operation are also returned to the client, and then thrown in the client as `UnknownUserException`. Non-Ice exceptions are returned to the client as `UnknownException`.

  The `cookie` out-parameter to `locate` allows you to return a local object to the object adapter. The object adapter does not care about the contents of that object (and it is legal to return a null cookie). Instead, the Ice run time passes whatever cookie you return from `locate` back to you when it calls `finished`. This allows you to pass an arbitrary amount of state from `locate` to the corresponding call to `finished`.

- `finished`

  If a call to `locate` has returned a servant to the Ice run time, the Ice run time dispatches the incoming request to the servant. Once the request is complete (that is, the operation being invoked has completed), the Ice run time calls `finished`, passing the servant whose operation has completed, the `Current` object for the request, and the cookie that was initially created by `locate`. This means that every call to `locate` is balanced by a corresponding call to `finished` (provided that `locate` actually returned a servant).

  If you throw an exception from `finished`, the Ice run time propagates the thrown exception back to the client. As for `locate`, you can throw user exceptions from `finished`. If a user exception is in the corresponding operation's exception specification, that user exception is returned to the client. User exceptions that are not in the corresponding operation's exception specification are also returned to the client, and then thrown by the client as `UnknownUserException`.

`finished` can also throw run-time exceptions. However, only `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException` are propagated without change to the client; other run-time exceptions are returned to the client as `UnknownLocalException`.

Non-Ice exceptions thrown from `finished` are returned to the client as `UnknownException`.

If both the operation implementation and `finished` throw a user exception, the exception thrown by `finished` overrides the exception thrown by the operation.

- `deactivate`

  The `deactivate` operation allows a servant locator to clean up once it is no longer needed. (For example, the locator might close a database connection.) The Ice run time passes the category of the servant locator being deactivated to the `deactivate` operation.

  The run time calls `deactivate` when destroying the object adapter to which the servant locator is attached. More precisely, `deactivate` is called when you call `destroy` on the object adapter, or when you call `destroy` on the communicator (which implicitly calls `destroy` on the object adapter).

  Once the run time has called `deactivate`, it is guaranteed that no further calls to `locate` or `finished` can happen, that is, `deactivate` is called exactly once, after all operations dispatched via this servant locator have completed.

  This also explains why `deactivate` is not called as part of `ObjectAdapter::deactivate`: `ObjectAdapter::deactivate` initiates deactivation and returns immediately, so it cannot call `ServantLocator::deactivate` directly, because there might still be outstanding requests dispatched via this servant locator that have to complete first — in turn, this would mean that either `ObjectAdapter::deactivate` could block (which it must not do) or that a call to `ServantLocator::deactivate` could be followed by one or more calls to `finished` (which must not happen either).

It is important to realize that the Ice run time does not "remember" the servant that is returned by a particular call to `locate`. Instead, the Ice run time simply dispatches an incoming request to the servant returned by `locate` and, once the request is complete, calls `finished`. In particular, if two requests for the same servant arrive more or less simultaneously, the Ice run time calls `locate` and `finished` once for each request. In other words, `locate` establishes the association between an object identity and a servant; that association is valid only for a single request and is never used by the Ice run time to dispatch a different request.

See Also

- The Active Servant Map
- The Current Object
- Run-Time Exceptions
- Object Adapter States

Threading Guarantees for Servant Locators

The Ice run time guarantees that every operation invocation that involves a servant locator is bracketed by calls to `locate` and `finished`, that is, every call to `locate` is balanced by a corresponding call to `finished` (assuming that the call to `locate` actually returned a servant, of course).

In addition, the Ice run time guarantees that `locate`, the operation, and `finished` are called by the same thread. This guarantee is important because it allows you to use `locate` and `finished` to implement thread-specific pre- and post-processing around operation invocations. (For example, you can start a transaction in `locate` and commit or roll back that transaction in `finished`, or you can acquire a lock in `locate` and release the lock in `finished`.

> Both transactions and locks usually are thread-specific, that is, only the thread that started a transaction can commit it or roll it back, and only the thread that acquired a lock can release the lock.

> If you are using asynchronous method dispatch, the thread that starts a call is not necessarily the thread that finishes it. In that case, `finished` is called by whatever thread executes the operation implementation, which may be a different thread than the one that called `locate`.

The Ice run time also guarantees that `deactivate` is called when you destroy the object adapter to which the servant locator is attached. The `deactivate` call is made only once all operations that involved the servant locator are finished, that is, `deactivate` is guaranteed not to run concurrently with `locate` or `finished`, and is guaranteed to be the last call made to a servant locator.

Beyond this, the Ice run time provides no threading guarantees for servant locators. In particular, it is possible for invocations of:

- `locate` to proceed concurrently (for the same object identity or for different object identities).
- `finished` to proceed concurrently (for the same object identity or for different object identities).
- `locate` and `finished` to proceed concurrently (for the same object identity or for different object identities).

These semantics allow you to extract the maximum amount of parallelism from your application code (because the Ice run time does not serialize invocations when serialization may not be necessary). Of course, this means that you must protect access to shared data from `locate` and `finished` with mutual exclusion primitives as necessary.

See Also

- The ServantLocator Interface
- The Ice Threading Model

Registering a Servant Locator

On this page:

Servant Locator Registration

An object adapter does not automatically know when you create a servant locator. Instead, you must explicitly register a servant locator with the object adapter:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        // ...

        void addServantLocator(ServantLocator locator, string category);

        ServantLocator removeServantLocator(string category);

        ServantLocator findServantLocator(string category);

        // ...
    }
}
```

As you can see, the object adapter allows you to add, remove, and find servant locators. Note that, when you register a servant locator, you must provide an argument for the `category` parameter. The value of the `category` parameter controls which object identities the servant locator is responsible for: only object identities with a matching `category` member trigger a corresponding call to `locate`. An incoming request for which no explicit entry exists in the active servant map (ASM) and with a category for which no servant locator is registered returns an `ObjectNotExistException` to the client.

`addServantLocator` has the following semantics:

- You can register exactly one servant locator for a specific category. Attempts to call `addServantLocator` for the same category more than once raise an `AlreadyRegisteredException`.
- You can register different servant locators for different categories, or you can register the same single servant locator multiple times (each time for a different category). In the former case, the category is implicit in the servant locator instance that is called by the Ice run time; in the latter case, the implementation of `locate` can find out which category the incoming request is for by examining the object identity member of the `Current` object that is passed to `locate`.
- It is legal to register a servant locator for the empty category. Such a servant locator is known as a *default servant locator*: if a request comes in for which no entry exists in the ASM, and whose category does not match the category of any other registered servant locator, the Ice run time calls `locate` on the default servant locator.

`removeServantLocator` removes and returns the servant locator for a specific category (including the empty category) with the following semantics:

- If no servant locator is registered for the specified category, the operation raises `NotRegisteredException`.
- Once a servant locator is successfully removed for the specified category, the Ice run time guarantees that no new incoming requests for that category are dispatched to the servant locator.
- A call to `removeServantLocator` returns immediately without waiting for the completion of any pending requests on that servant locator; such requests still complete normally by calling `finished` on the servant locator.
- Removing a servant locator does not cause Ice to invoke `deactivate` on that servant locator, as `deactivate` is only called when a registered servant locator's object adapter is destroyed.

`findServantLocator` allows you to retrieve the servant locator for a specific category (including the empty category). If no match is found, the operation returns null.

Call Dispatch Semantics for Servant Locators

The preceding rules may seem complicated, so here is a summary of the actions taken by the Ice run time to locate a servant for an incoming request.

Every incoming request implicitly identifies a specific object adapter for the request (because the request arrives at a specific transport endpoint and, therefore, identifies a particular object adapter). The incoming request carries an object identity that must be mapped to a servant. To locate a servant, the Ice run time goes through the following steps, in the order shown:

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.
2. If the category of the incoming object identity is non-empty, look for a default servant that is registered for that category. If such a default servant is registered, dispatch the request to that servant.
3. If the category of the incoming object identity is empty, or no default servant could be found for the category in step 2, look for a default servant that is registered for the empty category. If such a default servant is registered, dispatch the request to that servant.
4. If the category of the incoming object identity is non-empty and no servant could be found in the preceding steps, look for a servant locator that is registered for that category. If such a servant locator is registered, call `locate` on the servant locator and, if `locate` r eturns a servant, dispatch the request to that servant, followed by a call to `finished`; otherwise, if the call to `locate` returns null, raise `ObjectNotExistException` or `FacetNotExistException` in the client.
5. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 4, look for a default servant locator (that is, a servant locator that is registered for the empty category). If a default servant locator is registered, dispatch the request as for step 4.
6. Raise `ObjectNotExistException` or `FacetNotExistException` in the client. (`ObjectNotExistException` is raised if the ASM does not contain a servant with the given identity at all, `FacetNotExistException` is raised if the ASM contains a servant with a matching identity, but a non-matching facet.)

It is important to keep these call dispatch semantics in mind because they enable a number of powerful implementation techniques. Each technique allows you to streamline your server implementation and to precisely control the trade-off between performance, memory consumption, and scalability. To illustrate the possibilities, we will outline a number of the most common implementation techniques.

See Also

- Object Adapters
- Object Identity
- The Active Servant Map
- Default Servants
- Versioning
- Servant Locator Example

Servant Locator Example

To illustrate the servant locator concepts outlined so far, let us examine a (very simple) implementation. Consider that we want to create an electronic phone book for the entire world's telephone system (which, clearly, involves a very large number of entries, certainly too many to hold the entire phone book in memory). The actual phone book entries are kept in a large database. Also assume that we have a search operation that returns the details of a phone book entry. The Slice definitions for this application might look something like the following:

Slice

```
struct Details
{
    // Lots of details about the entry here...
}

interface PhoneEntry
{
    idempotent Details getDetails();
    idempotent void updateDetails(Details d);
    // ...
}

struct SearchCriteria
{
    // Fields to permit searching...
}

interface PhoneBook
{
    idempotent PhoneEntry* search(SearchCriteria c);
    // ...
}
```

The details of the application do not really matter here; the important point to note is that each phone book entry is represented as an interface for which we need to create a servant eventually, but we cannot afford to keep servants for all entries permanently in memory.

Each entry in the phone database has a unique identifier. This identifier might be an internal database identifier, or a combination of field values, depending on exactly how the database is constructed. The important point is that we can use this database identifier to link the proxy for an Ice object to its persistent state: we simply use the database identifier as the object identity. This means that each proxy contains the primary access key that is required to locate the persistent state of each Ice object and, therefore, instantiate a servant for that Ice object.

What follows is an outline implementation in C++. The class definition of our servant locator looks as follows:

C++11 C++98

```
class MyServantLocator : public Ice::ServantLocator
{
public:

    virtual std::shared_ptr<Ice::Object> locate(const Ice::Current&
current, std::shared_ptr<void>& cookie) override;

    virtual void finished(const Ice::Current& current, const
std::shared_ptr<Ice::Object>& servant, const std::shared_ptr<void>&
cookie) override;

    virtual void deactivate(const std::string&) = override;
};
```

```
class MyServantLocator : public Ice::ServantLocator
{
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& current,
Ice::LocalObjectPtr& cookie);

    virtual void finished(const Ice::Current& current,
const Ice::ObjectPtr& servant, const Ice::LocalObjectPtr& cookie);

    virtual void deactivate(const std::string& category);
};
```

Note that `MyServantLocator` inherits from `Ice::ServantLocator` and implements the pure virtual functions that are generated by the `slice2cpp` compiler for the `Ice::ServantLocator` interface. Of course, as always, you can add additional member functions, such as a constructor and destructor, and you can add private data members as necessary to support your implementation.

In C++, you can implement the `locate` member function along the following lines:

C++11 C++98

```
shared_ptr<Ice::Object>
MyServantLocator::locate(const Ice::Current& current, shared_ptr<void>&
cookie))
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    auto name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try
    {
        d = DB_lookup(name);
    }
    catch(const DB_error&)
    {
        return 0;
    }

    // We have the state, instantiate a servant and return it.
    //
    return make_shared<PhoneEntryI>(d);
}
```

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current& current,
Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try
    {
        d = DB_lookup(name);
    }
    catch(const DB_error&)
    {
        return 0;
    }

    // We have the state, instantiate a servant and return it.
    //
    return new PhoneEntryI(d);
}
```

For the time being, the implementations of `finished` and `deactivate` are empty and do nothing.

The `DB_lookup` call in the preceding example is assumed to access the database. If the lookup fails (presumably, because no matching record could be found), `DB_lookup` throws a `DB_error` exception. The code catches that exception and returns zero instead; this raises `ObjectNotExistException` in the client to indicate that the client used a proxy to a no-longer existent Ice object.

Note that `locate` instantiates the servant on the heap and returns it to the Ice run time. This raises the question of when the servant will be destroyed. The answer is that the Ice run time holds onto the servant for as long as necessary, that is, long enough to invoke the operation on the returned servant and to call `finished` once the operation has completed. Thereafter, the servant is no longer needed and the Ice run time destroys the smart pointer that was returned by `locate`. In turn, because no other smart pointers exist for the same servant, this causes the destructor of the `PhoneEntryI` instance to be called, and the servant to be destroyed.

The upshot of this design is that, for every incoming request, we instantiate a servant and allow the Ice run time to destroy the servant once the request is complete. Depending on your application, this may be exactly what is needed, or it may be prohibitively expensive — we will explore designs that avoid creation and destruction of a servant for every request shortly.

In Java, the implementation of our servant locator looks very similar:

Java Java Compat

```
import com.zeroc.Ice.ServantLocator;

public class MyServantLocator implements ServantLocator
{
    public ServantLocator.LocateResult locate(com.zeroc.Ice.Current cur
rent)
    {
        // Get the object identity. (We use the name member
as the database key.)
        String name = c.id.name;

        // Use the identity to retrieve the state from the database.
        //
        ServantDetails d;
        try
        {
            d = DB.lookup(name);
        }
        catch(DB.error e)
        {
            return new ServantLocator.LocateResult();
        }

        // We have the state, instantiate a servant and return it.
        //
        return new ServantLocator.LocateResult(new PhoneEntryI(d),
null);
    }

    public void finished(com.zeroc.Ice.Current current,
com.zeroc.Ice.Object servant, java.lang.Object cookie)
    {
    }

    public void deactivate(String category)
    {
    }
}
```

```
public class MyServantLocator implements Ice.ServantLocator
{
    public Ice.Object locate(Ice.Current current, Ice.LocalObjectHolder
cookie)
    {
        // Get the object identity. (We use the name member
        // as the database key.
        String name = c.id.name;

        // Use the identity to retrieve the state
        // from the database.
        //
        ServantDetails d;
        try
        {
            d = DB.lookup(name);
        }
        catch(DB.error e)
        {
            return null;
        }

        // We have the state, instantiate a servant and return it.
        //
        return new PhoneEntryI(d);
    }

    public void finished(Ice.Current current, Ice.Object servant,
java.lang.Object cookie)
    {
    }

    public void deactivate(String category)
    {
    }
}
```

The C# implementation is virtually identical to the Java implementation, so we do not show it here.

All implementations of `locate` follow the pattern illustrated by the previous pseudo-code:

1. Use the `id` member of the passed `Current` object to obtain the object identity. Typically, only the `name` member of the identity is used to retrieve servant state. The `category` member is normally used to select a servant locator. (We will explore use of the category member shortly.)
2. Retrieve the state of the Ice object from secondary storage (or the network) using the object identity as a key.
   - If the lookup succeeds, you have retrieved the state of the Ice object.
   - If the lookup fails, return null. In that case, the Ice object for the client's request truly does not exist, presumably, because that Ice object was deleted earlier, but the client still has a proxy to the now-deleted object.
3. Instantiate a servant and use the state retrieved from the database to initialize the servant. (In this example, we pass the retrieved state to the servant constructor.)

4. Return the servant.

Of course, before we can use our servant locator, we must inform the adapter of its existence prior to activating the adapter, for example (in Java or C#):

```
MyServantLocator sl = new MyServantLocator();
adapter.addServantLocator(sl, "");
```

Note that, in this example, we have installed the servant locator for the empty category. This means that `locate` on our servant locator will be called for invocations to any of our Ice objects (because the empty category acts as the default). In effect, with this design, we are not using the `category` member of the object identity. This is fine, as long as all our servants all have the same, single interface. However, if we need to support several different interfaces in the same server, this simple strategy is no longer sufficient.

See Also

- Servant Locators
- Object Identity
- Object Life Cycle
- The Current Object
- Using Identity Categories with Servant Locators

Using Identity Categories with Servant Locators

Our simple example always instantiates a servant of type `PhoneEntryI`. In other words, the servant locator implicitly is aware of the type of servant the incoming request is for. This is not a very realistic assumption for most servers because, usually, a server provides access to objects with several different interfaces. This poses a problem for our `locate` implementation: somehow, we need to decide inside `locate` what type of servant to instantiate. You have several options for solving this problem:

- Use a separate object adapter for each interface type and use a separate servant locator for each object adapter.

  This technique works fine, but has the down-side that each object adapter requires a separate transport endpoint, which is wasteful.

- Mangle a type identifier into the `name` component of the object identity.

  This technique uses part of the object identity to denote what type of object to instantiate. For example, in our file system application, we have directory and file objects. By convention, we could prepend a 'd' to the identity of every directory and prepend an 'f' to the identity of every file. The servant locator then can use the first letter of the identity to decide what type of servant to instantiate:

  C++11 C++98

```
std::shared_ptr<Ice::Object>
MyServantLocator::locate(const Ice::Current& current,
std::shared_ptr<void>& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    auto name = c.id.name;
    auto realId = c.id.name.substr(1);
    try
    {
        if(name[0] == 'd')
        {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return std::make_shared<DirectoryI>(d);
        }
        else
        {
            // The request is for a file.
            //
            FileDetails d = DB_lookup(realId);
            return std::make_shared<FileI>(d);
        }
    }
    catch(DatabaseNotFoundException&)
    {
        return 0;
    }
}
```

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current& current,
Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;
    std::string realId = c.id.name.substr(1);
    try
    {
        if(name[0] == 'd')
        {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return new DirectoryI(d);
        }
        else
        {
            // The request is for a file.
            //
            FileDetails d = DB_lookup(realId);
            return new FileI(d);
        }
    }
    catch(DatabaseNotFoundException&)
    {
        return 0;
    }
}
```

While this works, it is awkward: not only do we need to parse the `name` member to work out what type of object to instantiate, but we also need to modify the implementation of `locate` whenever we add a new type to our application.

- Use the `category` member of the object identity to denote the type of servant to instantiate.

  This is the recommended approach: for every interface type, we assign a separate identifier as the value of the `category` member of the object identity. (For example, we can use 'd' for directories and 'f' for files.) Instead of registering a single servant locator, we create two different servant locator implementations, one for directories and one for files, and then register each locator for the appropriate category:

  C++11 C++98

```
class DirectoryLocator : public Ice::ServantLocator
{
public:

    virtual std::shared_ptr<Ice::Object> locate(const Ice::Current
& current, std::shared_ptr<void>& cookie) override
    {
        // Code to locate and instantiate a directory here...
    }

    virtual void finished(const Ice::Current& current,
const std::shared_ptr<Ice::Object>& servant,
const std::shared_ptr<void>& cookie) override
    {
    }

    virtual void deactivate(const std::string& category) override
    {
    }
};

class FileLocator : public Ice::ServantLocator
{
public:

    virtual std::shared_ptr<Ice::Object> locate(const Ice::Current
& current, std::shared_ptr<void>& cookie) override
    {
        // Code to locate and instantiate a file here...
    }

    virtual void finished(const Ice::Current& current,
const std::shared_ptr<Ice::Object> servant,
const std::shared_ptr<void>& cookie) override
    {
    }

    virtual void deactivate(const std::string& category) override
    {
    }
};

// ...

// Register two locators, one for directories and
// one for files.
//
adapter->addServantLocator(std::make_shared<DirectoryLocator>(), "
d");
adapter->addServantLocator(std::make_shared<FileLocator>(), "f");
```

```
class DirectoryLocator : public Ice::ServantLocator
{
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& current,
Ice::LocalObjectPtr& cookie)
    {
        // Code to locate and instantiate a directory here...
    }

    virtual void finished(const Ice::Current& current,
const Ice::ObjectPtr& servant, const Ice::LocalObjectPtr& cookie)
    {
    }

    virtual void deactivate(const std::string& category)
    {
    }
};

class FileLocator : public Ice::ServantLocator
{
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& current, Ice
::LocalObjectPtr& cookie)
    {
        // Code to locate and instantiate a file here...
    }

    virtual void finished(const Ice::Current& current,
const Ice::ObjectPtr& servant, const Ice::LocalObjectPtr& cookie)
    {
    }

    virtual void deactivate(const std::string& category)
    {
    }
};

// ...

// Register two locators, one for directories and
// one for files.
//
adapter->addServantLocator(new DirectoryLocator(), "d");
adapter->addServantLocator(new FileLocator(), "f");
```

- Yet another option is to use the `category` member of the object identity, but to use a single default servant locator (that is, a locator for the empty category). With this approach, all invocations go to the single default servant locator, and you can switch on the `category` value inside the implementation of the `locate` operation to determine which type of servant to instantiate. However, this approach is harder to maintain than the previous one; the `category` member of the Ice object identity exists specifically to support servant locators, so you might as well use it as intended.

See Also

- Servant Locator Example
- Object Identity

Using Cookies with Servant Locators

Occasionally, it can be useful for a servant locator to pass information between `locate` and `finished`. For example, the implementation of `locate` could choose among a number of alternative database backends, depending on load or availability and, to properly finalize state, the implementation of `finished` might need to know which database was used by `locate`. To support such scenarios, you can create a cookie in your `locate` implementation; the Ice run time passes the value of the cookie to `finished` after the operation invocation has completed. The cookie must derive from `Ice::LocalObject` and can contain whatever state and member functions are useful to your implementation:

C++11 C++98

```
class MyCookie
{
public:
    // Whatever is useful here...
};

class MyServantLocator : public Ice::ServantLocator
{
public:

    virtual std::shared_ptr<Ice::Object>
locate(const Ice::Current& current, std::shared_ptr<void>& cookie)
override
    {
        // Code as before...

        // Allocate and initialize a cookie.
        //
        cookie = std::make_shared<MyCookie>(...);

        return std::make_shared<PhoneEntryI>();
    }

    virtual void finished(const Ice::Current& current,
const std::shared_ptr<Ice::Object>& servant,
const std::shared_ptr<void>& cookie) override
    {
        // Down-cast cookie to actual type.
        //
        auto mc = std::static_pointer_cast<MyCookie>(cookie);

        // Use information in cookie to clean up...
        //
        // ...
    }

    virtual void deactivate(const std::string& category) override
    {
    }
};
```

```
class MyCookie : public Ice::LocalObject
{
public:
    // Whatever is useful here...
};
typedef IceUtil::Handle<MyCookie> MyCookiePtr;

class MyServantLocator : public Ice::ServantLocator
{
public:

    virtual Ice::ObjectPtr locate(const Ice::Current& current,
Ice::LocalObjectPtr& cookie)
    {
        // Code as before...

        // Allocate and initialize a cookie.
        //
        cookie = new MyCookie(...);

        return new PhoneEntryI;
    }

    virtual void finished(const Ice::Current& current,
const Ice::ObjectPtr& servant, const Ice::LocalObjectPtr& cookie)
    {
        // Down-cast cookie to actual type.
        //
        MyCookiePtr mc = MyCookiePtr::dynamicCast(cookie);

        // Use information in cookie to clean up...
        //
        // ...
    }

    virtual void deactivate(const std::string& category)
    {
    }
};
```

See Also

- Servant Locators

Default Servants

On this page:

Overview of Default Servants

The Active Servant Map (ASM) is a simple lookup table that maintains a one-to-one mapping between object identities and servants. Although the ASM is easy to understand and offers efficient indexing, it does not scale well when the number of objects is very large. Scalability is a common problem with object-oriented middleware: servers frequently are used as front ends to large databases that are accessed remotely by clients. The server's job is to present an object-oriented view to clients of a very large number of records in the database. Typically, the number of records is far too large to instantiate servants for even a fraction of the database records.

A common technique for solving this problem is to use *default servants*. A default servant is a servant that, for each request, takes on the persona of a different Ice object. In other words, the servant changes its behavior according to the object identity that is accessed by a request, on a per-request basis. In this way, it is possible to allow clients access to an unlimited number of Ice objects with only a single servant in memory. A default servant is essentially a specialized version of a servant locator that satisfies the majority of use cases with a simpler API, whereas a servant locator provides more flexibility for those applications that require it.

Default servant implementations are attractive not only because of the memory savings they offer, but also because of the simplicity of implementation: in essence, a default servant is a facade [1] to the persistent state of an object in the database. This means that the programming required to implement a default servant is typically minimal: it simply consists of the code required to read and write the corresponding database records.

A default servant is a regular servant that you implement and register with an object adapter. For each incoming request, the object adapter first attempts to locate a servant in its ASM. If no servant is found, the object adapter dispatches the request to a default servant. With this design, a default servant is the object adapter's servant of last resort if no match was found in the ASM.

Implementing a default servant requires a somewhat different mindset than the typical "one servant per Ice object" strategy used in less advanced applications. The most important quality of a default servant is its statelessness: it must be prepared to dispatch multiple requests simultaneously for different objects. The price we have to pay for the unlimited scalability and reduced memory footprint is performance: default servants typically make a database access for every invoked operation, which is obviously slower than caching state in memory as part of a servant that has been added to the ASM. However, this does not mean that default servants carry an unacceptable performance penalty: databases often provide sophisticated caching, so even though the operation implementations read and write the database, as long as they access cached state, performance may be entirely acceptable.

Default Servant API

The default servant API consists of the following operations in the object adapter interface:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        void addDefaultServant(Object servant, string category);
        Object removeDefaultServant(string category);
        Object findDefaultServant(string category);

        // ...
    }
}
```

As you can see, the object adapter allows you to add, remove, and find default servants. Note that, when you register a default servant, you must provide an argument for the `category` parameter. The value of the `category` parameter controls which object identities the default servant is responsible for: only object identities with a matching `category` member trigger a dispatch to this default servant. An incoming request for which no explicit entry exists in the ASM and with a category for which no default servant is registered returns an `ObjectNotExistException` to the client.

`addDefaultServant` has the following semantics:

- You can register exactly one default servant for a specific category. Attempts to call `addDefaultServant` for the same category more than once raise an `AlreadyRegisteredException`.
- You can register different default servants for different categories, or you can register the same single default servant multiple times (each time for a different category). In the former case, the category is implicit in the default servant instance that is called by the Ice run time; in the latter case, the servant can find out which category the incoming request is for by examining the object identity member of the `Current` object that is passed to the dispatched operation.
- It is legal to register a default servant for the empty category. Such a servant is used if a request comes in for which no entry exists in the ASM, and whose category does not match the category of any other registered default servant.

`removeDefaultServant` removes the default servant for the specified category. Attempts to remove a non-existent default servant raise `NotRegisteredException`. The operation returns the removed default servant. Once a default servant is successfully removed for the specified category, the Ice run time guarantees that no new incoming requests for that category are dispatched to the servant.

The `findDefaultServant` operation allows you to retrieve the default servant for a specific category (including the empty category). If no default servant is registered for the specified category, `findDefaultServant` returns null.

Threading Guarantees for Default Servants

The threading semantics for a default servant are no different than for a servant registered in the ASM: operations may be dispatched on a default servant concurrently, for the same object identity or for different object identities. If you have configured the communicator with multiple dispatch threads, your default servant must protect access to shared data with appropriate locks.

Call Dispatch Semantics for Default Servants

This section summarizes the actions taken by the Ice run time to locate a servant for an incoming request.

Every incoming request implicitly identifies a specific object adapter for the request (because the request arrives at a specific transport endpoint and, therefore, identifies a particular object adapter). The incoming request carries an object identity that must be mapped to a servant. To locate a servant, the Ice run time goes through the following steps, in the order shown:

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.
2. If the category of the incoming object identity is non-empty, look for a default servant that is registered for that category. If such a default servant is registered, dispatch the request to that servant.
3. If the category of the incoming object identity is empty, or no default servant could be found for the category in step 2, look for a default servant that is registered for the empty category. If such a default servant is registered, dispatch the request to that servant.
4. If the category of the incoming object identity is non-empty and no servant could be found in the preceding steps, look for a servant locator that is registered for that category. If such a servant locator is registered, call `locate` on the servant locator and, if `locate` r

eturns a servant, dispatch the request to that servant, followed by a call to `finished`; otherwise, if the call to `locate` returns null, raise `ObjectNotExistException` or `FacetNotExistException` in the client. (`ObjectNotExistException` is raised if the ASM does not contain a servant with the given identity at all, `FacetNotExistException` is raised if the ASM contains a servant with a matching identity, but a non-matching facet.)

5. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 4, look for a default servant locator (that is, a servant locator that is registered for the empty category). If a default servant locator is registered, dispatch the request as for step 4.
6. Raise `ObjectNotExistException` or `FacetNotExistException` in the client. (`ObjectNotExistException` is raised if the ASM does not contain a servant with the given identity at all, `FacetNotExistException` is raised if the ASM contains a servant with a matching identity, but a non-matching facet.)

It is important to keep these call dispatch semantics in mind because they enable a number of powerful implementation techniques. Each technique allows you to streamline your server implementation and to precisely control the trade-off between performance, memory consumption, and scalability.

Guidelines for Implementing Default Servants

This section provides some guidelines to assist you in implementing default servants effectively.

Object Identity is the Key

When an incoming request is dispatched to the default servant, the target object identity is provided in the `Current` argument. The `name` fiel d of the identity typically supplies everything the default servant requires in order to satisfy the request. For instance, it may serve as the key in a database query, or even hold an encoded structure in some proprietary format that your application uses to convey more than just a string.

Naturally, the client can also pass arguments to the operation that assist the default servant in retrieving whatever state it requires. However, this approach can easily introduce implementation artifacts into your Slice interfaces, and in most cases the client should not need to know that the server is implemented with a default servant. If at all possible, use only the object identity.

Minimize Contention

For better scalability, the default servant's implementation should strive to eliminate contention among the dispatch threads. As an example, when a database holds the default servant's state, each of the servant's operations usually begins with a query. Assuming that the database API is thread-safe, the servant needs to perform no explicit locking of its own. With a copy of the state in hand, the implementation can work with function-local data to satisfy the request.

Combine Strategies

The ASM still plays a useful role even in applications that are ideally suited for default servants. For example, there is no need to implement a singleton object as a default servant: if there can only be one instance of the object, implementing it as a default servant does nothing to improve your application's scalability.

Applications often install a handful of servants in the ASM while servicing the majority of requests in a default servant. For example, a database application might install a singleton query object in the ASM while using a default servant to process all invocations on the database records.

Categories Denote Interfaces

In general, all of the objects serviced by a default servant must have the same interface. If you only need a default servant for one interface, you can register the default servant with an empty category string. However, to implement several interfaces, you will need a default servant implementation for each one. Furthermore, you must take steps to ensure that the object adapter dispatches an incoming request to the appropriate default servant. The `category` field of the object identity is intended to serve this purpose.

For example, a process control system might have interfaces named `Sensor` and `Switch`. To direct requests to the proper default servant, the application uses the symbol `Sensor` or `Switch` as the category of each object's identity, and registers corresponding default servants having those same categories with the object adapter.

Plan for the Future

If you suspect that you might eventually need to implement more than one interface with default servants, we recommend using a non-empty category even if you start out having only one default servant. Adding another default servant later becomes much easier if the application is already designed to operate correctly with categories.

Throw exceptions

If a request arrives for an object that no longer exists, it is the default servant's responsibility to raise `ObjectNotExistException` to properly manage object life cycles.

Handle `ice_ping`

One issue you need to be aware of with default servants is the need to override `ice_ping`: the default implementation of `ice_ping` that the servant inherits from its skeleton class always succeeds. For servants that are registered with the ASM, this is exactly what we want; however, for default servants, `ice_ping` must fail if a client uses a proxy to an Ice object that no longer exists. To avoid getting successful `ice_ping` invocations for non-existent Ice objects, you must override `ice_ping` in the default servant. The implementation must check whether the object identity for the request denotes a still-existing Ice object and, if not, raise `ObjectNotExistException`.

It is good practice to override `ice_ping` if you are using default servants. Because you cannot override operations on `Ice::Object` using a Java or C# tie servant (or an Objective-C delegate servant), you must implement default servants by deriving from the generated skeleton class if you choose to override `ice_ping`.

Consider Interceptors

A dispatch interceptor is often installed as a default servant.

Use a `Blobject` Default Servant to Forward Messages

Message forwarding services, such as Glacier2, can be implemented simply and efficiently with a `Blobject` default servant. Such a servant simply chooses a destination to forward a request to, without decoding any of the parameters.

See Also

- The Active Servant Map
- Object Identity
- Servant Locators
- Object Adapters
- The Current Object
- The Ice Threading Model
- Versioning
- Dispatch Interceptors
- Dynamic Invocation and Dispatch Overview
- Glacier2

References

1. Gamma, E., et al. 1994. Design Patterns. Reading, MA: Addison-Wesley.

Dispatch Interceptors

A dispatch interceptor is a server-side mechanism that allows you to intercept incoming client requests before they are given to a servant. The interceptor can examine the incoming request and in particular its `Current` information.

A dispatch interceptor can dispatch a request to a servant and check whether the dispatch was successful; if not, the interceptor can choose to retry the dispatch. This functionality is useful to automatically retry requests that have failed due to a recoverable error condition, such as a database deadlock exception.

Dispatch Interceptors are currently available in C++11, C++98, C#, Java, Java Compat and Objective-C. They are defined through a native API in each programming language.

On this page:

Dispatch Interceptor Position

A dispatch interceptor is a servant that delegates requests to another servant. This other servant can be the real servant that unmarshals the parameters carried by the request and implements the target operation. It could also be another dispatch interceptor: this way, you can create a chain of dispatch interceptors, with the real servant at the end of this chain.

You register a dispatch interceptor with an object adapter like any other servant: it can be inserted into this object adapter's Active Servant Map, or registered as a default servant, or returned by a servant locator.

> Dispatch interceptors are most often registered as default servants.

A dispatch follows these steps:

1. The Ice run time reads the incoming request using a thread from its server thread pool (or object adapter thread pool if one is configured, or client thread pool for requests over bidir connections).
2. If a dispatcher is configured, the Ice run time gives the request to the dispatcher to allow the dispatcher to execute this request in a different thread.
3. The Ice run time locates the target servant (this may call `locate` on a servant locator).
4. Assuming the target servant is a dispatch interceptor, it calls `ice_dispatch` on the next servant (which could be another dispatch interceptor); eventually a dispatch interceptor calls `ice_dispatch` on the real servant.
5. The real servant unmarshals the input parameters, executes the operation and then:
   - marshals the return and out parameters,
   - throws an exception,
   - or does not complete synchronously (if it's implemented using AMD).
6. The dispatch interceptor(s) now have the opportunity to execute code "on the way out", after completion of the dispatch by the real servant (with AMD, only the synchronous part of this dispatch has completed).
7. For a synchronous dispatch through a servant locator, the Ice run time calls `finished` on the servant locator.
8. For a synchronous dispatch, the Ice run time sends the reply to the client: the return and out parameters previously marshaled by the real servant, or an exception caught and marshaled by the Ice run time.

A dispatch interceptor can also loop through steps 4 and 6 and dispatch several times the same request before returning the result to Ice. For example:

C++11

```
bool
MyDispatchInterceptor::dispatch(Ice::Request& request)
{
    for(;;)
    {
        try
        {
            TransactionHolder tx(_db);
            bool sync =
findServant(request.getCurrent().id)->ice_dispatch(request);
            assert(sync); // this example works only for synchronous
dispatch
            tx.commit();
            return sync;
        }
        catch(const DeadlockException&)
        {
            // Run this dispatch again, with the same request
            //
        }
    }
}
```

Dispatch Interceptor API

You create a dispatch interceptor by providing a class that derives from the `DispatchInterceptor` abstract class and implements the `dispatch` function or method. The job of `dispatch` is to pass the request to the servant and to return the value returned by the servant's `ice_dispatch` call.

The Ice run time provides basic information about the request to the `dispatch` function in the form of a `Request` object:

C++11 C++98 C# Java Java CompatObjC

```
namespace Ice
{
    class Request
    {
    public:
        virtual ~Request();
        virtual const Current& getCurrent() = 0;
    };
}
```

```
namespace Ice
{
    class Request
    {
    public:
        virtual ~Request();
        virtual const Current& getCurrent() = 0;
    };
}
```

```
namespace Ice
{
    public interface Request
    {
        Current getCurrent();
    }
}
```

```
package com.zeroc.Ice;

public interface Request
{
    Current getCurrent();
}
```

```
package Ice;

public interface Request
{
    Current getCurrent();
}
```

```
@protocol ICERequest <NSObject>
-(ICECurrent*) getCurrent;
@end
```

`getCurrent` provides access to the `Current` object for the request, which provides access to information about the request, such as the object identity of the target object, the object adapter used to dispatch the request, and the operation name.

Note that `Request`, for performance reasons, is *not* thread-safe. This means that you must not concurrently dispatch from different threads using the same `Request` object. (Concurrent dispatch for different requests does not cause any problems.). We also recommend that you do not change thread in your dispatch interceptor as this would defeat the dispatcher mechanism.

C++11 C++98 C# Java Java Compat Obj-C

```
namespace Ice
{
    class DispatchInterceptor : public virtual Object
    {
    public:
        virtual bool dispatch(Request&) = 0;
    };
}
```

The `dispatch` return value is a bool that indicates whether dispatch was executed synchronously (`true`) or asynchronously with AMD (`false`). You must return the value returned by the servant `ice_dispatch` function, or throw an exception.

```
namespace Ice
{
    class DispatchInterceptor : public virtual Object
    {
    public:
         virtual bool dispatch(Request&) = 0;
    };
    typedef IceInternal::Handle<DispatchInterceptor>
DispatchInterceptorPtr;
}
```

The `dispatch` return value is a bool that indicates whether dispatch was executed synchronously (`true`) or asynchronously with AMD (`false`). You must return the value returned by the servant `ice_dispatch` function, or throw an exception.

```
namespace Ice
{
    public abstract class DispatchInterceptor : Ice.ObjectImpl
    {
        public abstract System.Threading.Tasks.Task<Ice.OutputStream>
dispatch(Request request);
    }
}
```

The `dispatch` return value is a `Task` instance if the request was dispatched asynchronously with AMD, or `null` if the request was dispatched synchronously. You must return the same value as returned by the servant `ice_dispatch` method, or a continuation created from the returned `Task`, or throw an exception. You can check for the completion of the `Task` and retry a dispatch if for example the task completed with an exception. It is fine to abandon a returned `Task` to retry a dispatch.

> If the servant implementation uses the `async` keyword for the dispatch of a request, `ice_dispatch` always returns a `Task` object even if the dispatch completed synchronously with a response or exception.

```
package com.zeroc.Ice;

import java.util.concurrent.CompletionStage;

public abstract class DispatchInterceptor implements
com.zeroc.Ice.Object
{
    public abstract CompletionStage<OutputStream> dispatch(Request
request) throws UserException;
    ...
}
```

The `dispatch` return value is a `CompletionStage<OutputStream>` if the request was dispatched asynchronously with AMD, or `null` if the request was dispatched synchronously. You must return the value returned by the servant `ice_dispatch` method, or a dependent `ContinuationStage`, or throw an exception. You can check for the completion of the `CompletionStage` and retry a dispatch if for example the request completed with an exception. It is fine to abandon a returned `CompletionStage` to retry a dispatch.

```
package Ice;

public abstract class DispatchInterceptor extends ObjectImpl
{
    public abstract boolean dispatch(Request request) throws
Ice.UserException;
}
```

The `dispatch` return value is a boolean that indicates whether dispatch was executed synchronously (`true`) or asynchronously with AMD (`false`). You must return the value returned by the servant `ice_dispatch` function or throw an exception.

```
@protocol ICEDispatchInterceptor <ICEObject>
-(void) dispatch:(id<ICERequest>)request;
@end
```

The Objective-C mapping does not support asynchronous dispatch (AMD), therefore the return type of the `dispatch` method is void.

Using a Dispatch Interceptor

Your implementation of the `dispatch` function or method must dispatch the request to the actual servant. Here is a very simple example implementation of an interceptor that dispatches the request to the servant passed to the interceptor's constructor:

C++11 C++98

```
class InterceptorI : public Ice::DispatchInterceptor
{
public:
    InterceptorI(std::shared_ptr<Ice::Object> servant) :
        _servant(std::move(servant))
    {
    }

    virtual bool dispatch(Ice::Request& request) override
    {
        return _servant->ice_dispatch(request);
    }

    std::shared_ptr<Ice::Object> _servant;
};
```

```
class InterceptorI : public Ice::DispatchInterceptor
{
public:
    InterceptorI(const Ice::ObjectPtr& servant) :
        _servant(servant)
    {
    }

    virtual bool dispatch(Ice::Request& request)
    {
        return _servant->ice_dispatch(request);
    }

    Ice::ObjectPtr _servant;
};
```

Note that our implementation of `dispatch` calls `ice_dispatch` on the target servant to dispatch the request. `ice_dispatch` does the work of actually invoking the operation. Also note that `dispatch` returns whatever is returned by `ice_dispatch`.

We can use this interceptor to intercept requests to a servant of any type as follows:

C++11 C++98

```
auto servant = std::make_shared<ExampleI>();
auto interceptor = std::make_shared<InterceptorI>(std::move(servant));
// give the servant to the interceptor
adapter->add(interceptor, Ice::stringToIdentity("ExampleServant"));
```

```
ExampleIPtr servant = new ExampleI;
Ice::DispatchInterceptorPtr interceptor = new InterceptorI(servant);
adapter->add(interceptor, Ice::stringToIdentity("ExampleServant"));
```

Dispatch Interceptor with Asynchronous Method Dispatch (AMD)

When the real servant dispatches a request asynchronously, the result (or exception) is typically not available "on the way out" in the dispatching thread. This creates two challenges:

- The dispatch interceptor may want to be notified when the dispatch completes.
- The dispatch interceptor may want to retry the initial dispatch and prevent earlier dispatch attempts from returning a response or exception to the client.

The solution depends on the language mapping:

Using Callbacks (C++11, C++98, Java Compat)

With C++11, `ice_dispatch` on servants accepts two optional function parameters:

C++11

```
namespace Ice
{
    class Object
    {
    public:
         virtual bool ice_dispatch(Ice::Request& request,
                                   std::function<bool()> response =
nullptr,

std::function<bool(std::exception_ptr)> error = nullptr);
         ...

    };
}
```

When a servant dispatches a request asynchronously, `ice_dispatch` returns false and then `response` or `error` is called when the dispatch completes. Both `response` and `error` (that you supply) return a `bool` parameter that indicates whether Ice should send the response or exception to the client (when the return value is true), or just ignore this result (when the return value is false). A null `response` or `error` function is equivalent to a function that returns true.

> Occasionally, you may attempt to re-dispatch a request that has already completed–meaning Ice has already sent the response or exception from a previous attempt back to the client. In this case, your dispatch interceptor will receive a `ResponseSentExcepti on`: it should then rethrow this exception or return true (like for a synchronous dispatch).

With C++98 and Java Compat, `ice_dispatch` on servants accepts a comparable `DispatchInterceptorAsyncCallback` parameter:

C++98 Java Compat

```
namespace Ice
{
    class DispatchInterceptorAsyncCallback : public virtual
IceUtil::Shared
    {
    public:

        virtual ~DispatchInterceptorAsyncCallback();

        virtual bool response() = 0;
        virtual bool exception(const std::exception&) = 0;
        virtual bool exception() = 0;
    };
    typedef IceUtil::Handle<DispatchInterceptorAsyncCallback>
DispatchInterceptorAsyncCallbackPtr;

    class Object : public virtual IceUtil::Shared
    {
    public:
         virtual bool ice_dispatch(Ice::Request& request, const
DispatchInterceptorAsyncCallbackPtr& = 0);
         ...
    };
}
```

```
package Ice;
public interface DispatchInterceptorAsyncCallback
{
    boolean response();
    boolean exception(java.lang.Exception ex);
}

public interface Object
{
    boolean ice_dispatch(Request request,
DispatchInterceptorAsyncCallback cb)
        throws Ice.UserException;

    boolean ice_dispatch(Request request)
        throws Ice.UserException;

    ...
}
```

Using Futures (C# and Java)

In C# and Java, `ice_dispatch` returns a non-null `Task<Ice.OutputStream>` (C#) or `CompletionStage<OutputStream>` (Java) for asynchronous dispatches:

C# Java

```
namespace Ice
{
    public interface Object : ICloneable
    {
        Task<OutputStream> ice_dispatch(Request request);
        ....
    }
}
```

```
package com.zeroc.Ice;

public interface Object
{
    CompletionStage<OutputStream> ice_dispatch(Request request)
        throws Ice.UserException;

    ...
}
```

Your dispatch interceptor can be notified of the completion of an asynchronous dispatch by registering an action with this `Task` or `CompletionStage` (using `ContinueWith` in C# or for example `whenComplete` in Java).

Your dispatch interceptor can also forward several times the same request to the real servant and only return the `Task` or `CompletionStage` provided by the "good" dispatch back to Ice; the `Task` or `CompletionStage` returned by the other dispatches are simply discarded.

See Also

1. The Current Object
2. The Active Servant Map
3. Servant Locators
4. Default Servants

Client-Server Features

This section presents APIs and features best understood while considering both sides of a client-server interaction.

Client-specific and server-specific features are presented in separate sections, Client-Side Features and Server-Side Features.

Topics

The Ice Threading Model

Ice is inherently a multi-threaded platform. There is no such thing as a single-threaded server in Ice. As a result, you must concern yourself with concurrency issues: if a thread reads a data structure while another thread updates the same data structure, havoc will ensue unless you protect the data structure with appropriate locks. In order to build Ice applications that behave correctly, it is important that you understand the threading semantics of the Ice run time. Here we discuss Ice's *thread pool* concurrency model and provide guidelines for writing thread-safe Ice applications.

Topics

Thread Pools

A thread pool is a collection of threads that the Ice run time draws upon to perform specific tasks.

On this page:

Introduction to Thread Pools

Each communicator creates two thread pools:

- The *client thread pool* services outgoing connections, which primarily involves handling the replies to outgoing requests and includes notifying AMI callback objects. If a connection is used in bidirectional mode, the client thread pool also dispatches incoming callback requests.
- The *server thread pool* services incoming connections. It dispatches incoming requests and, for bidirectional connections, processes replies to outgoing requests.

By default, these two thread pools are shared by all of the communicator's object adapters. If necessary, you can configure individual object adapters to use a private thread pool instead.

If a thread pool is exhausted because all threads are currently dispatching a request, additional incoming requests are transparently delayed until a request completes and relinquishes its thread; that thread is then used to dispatch the next pending request. Ice minimizes thread context switches in a thread pool by using a leader-follower implementation [1].

Configuring Thread Pools

Each thread pool has a unique name that serves as the prefix for its configuration properties:

- *name*`.Size`
  This property specifies the initial size of the thread pool. If not defined, the default value is 1.
- *name*`.SizeMax`
  This property specifies the maximum size of the thread pool. If not defined, the default value is 1. If the value of this property is less than that of *name*`.Size`, this property is adjusted to be equal to *name*`.Size`.
- *name*`.SizeWarn`
  This property sets a high water mark; when the number of threads in a pool reaches this value, the Ice run time logs a warning message. If you see this warning message frequently, it could indicate that you need to increase the value of *name*`.SizeMax`. The default value is 0, which disables the warning.
- *name*`.StackSize`
  This property specifies the number of bytes to use as the stack size of threads in the thread pool. The operating system's default is used if this property is not defined or is set to 0.
- *name*`.Serialize`
  Setting this property to a value greater than 0 forces the thread pool to serialize all messages received over a connection. It is unnecessary to enable serialization for a thread pool whose maximum size is 1 because such a thread pool is already limited to processing one message at a time. For thread pools with more than one thread, serialization can have a negative impact on latency and throughput. If not defined, the default value is 0. We discuss this feature in more detail in Thread Pool Design Considerations.
- *name*`.ThreadIdleTime`
  This property specifies the number of seconds that a thread in the thread pool must be idle before it terminates. The default value is 60 seconds if this property is not defined. Setting it to 0 disables the termination of idle threads.

For configuration purposes, the names of the client and server thread pools are `Ice.ThreadPool.Client` and `Ice.ThreadPool.Server`, respectively. As an example, the following properties establish the initial and maximum sizes for these thread pools:

```
Ice.ThreadPool.Client.Size=1
Ice.ThreadPool.Client.SizeMax=10
Ice.ThreadPool.Server.Size=1
Ice.ThreadPool.Server.SizeMax=10
```

To monitor the thread pool activities of the Ice run time, you can enable the `Ice.Trace.ThreadPool` property. Setting this property to a non-0 value causes the Ice run time to log a message when it creates a thread pool, as well as each time the size of a thread pool increases or decreases.

Dynamic Thread Pools

A *dynamic* thread pool can grow and shrink when necessary in response to changes in an application's work load. All thread pools have at least one thread, but a dynamic thread pool can grow as the demand for threads increases, up to the pool's maximum size. Threads may also be terminated automatically when they have been idle for some time.

The dynamic nature of a thread pool is determined by the configuration properties *name*`.Size`, *name*`.SizeMax`, and *name*`.ThreadIdleTime`. A thread pool is not dynamic in its default configuration because *name*`.Size` and *name*`.SizeMax` are both set to 1, meaning the pool can never grow to contain more than a single thread. To configure a dynamic thread pool, you must set at least one of *name*`.Size` or *name*`.SizeMax` to a value greater than 1. We can use several configuration scenarios to explore the semantics of dynamic thread pools in greater detail:

- *name*`.SizeMax=5`

  This thread pool initially contains a single thread because *name*`.Size` has a default value of 1, and Ice can grow the pool up to the maximum of 5 threads. During periods of inactivity, idle threads terminate after 60 seconds (the default value for *name*`.ThreadIdleTime`) until the pool contains just 1 thread again.

- *name*`.Size=3`
  name`.SizeMax=5`

  This thread pool starts with 3 active threads but otherwise behaves the same as in the previous configuration. The pool can still shrink to a size of 1 as threads become idle.

- *name*`.Size=3`
  name`.ThreadIdleTime=10`

  This thread pool starts with 3 active threads and shrinks quickly to 1 thread during periods of inactivity. As demand increases again, the thread pool can return to its maximum size of 3 threads (*name*`.SizeMax` defaults to the value of *name*`.Size`).

- *name*`.SizeMax=5`
  name`.ThreadIdleTime=0`

  This thread pool can grow from its initial size of 1 thread to contain up to 5 threads, but it will never shrink because *name*`.ThreadIdleTime` is set to 0.

- *name*`.Size=5`
  name`.ThreadIdleTime=0`

  This thread pool starts with 5 threads and can neither grow nor shrink.

To summarize, the value of *name*`.ThreadIdleTime` determines whether (and how quickly) a thread pool can shrink to a size of 1. A thread pool that shrinks can also grow to its maximum size. Finally, setting *name*`.SizeMax` to a value larger than *name*`.Size` allows a thread pool to grow beyond its initial capacity.

See Also

- Thread Pool Design Considerations
- Bidirectional Connections
- Object Adapters
- Object Adapter Thread Pools
- Ice.ThreadPool.*

References

1. Schmidt, D. C. et al. 2000. "Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching". In *Proceedings of the 7th Pattern Languages of Programs Conference*, WUCS-00-29, Seattle, WA: University of Washington.

Object Adapter Thread Pools

The default behavior of an object adapter is to share the thread pools of its communicator and, for many applications, this behavior is entirely sufficient. However, the ability to configure an object adapter with its own thread pool is useful in certain situations:

- When the concurrency requirements of an object adapter does not match those of its communicator. In a server with multiple object adapters, the configuration of the communicator's client and server thread pools may be a good match for some object adapters, but others may have different requirements. For example, the servants hosted by one object adapter may not support concurrent access, in which case limiting that object adapter to a single-threaded pool eliminates the need for synchronization in those servants. On the other hand, another object adapter might need a multi-threaded pool for better performance.
- To ensure that a minimum number of threads is available for dispatching requests to an adapter's servants. This is especially important for eliminating the possibility of deadlocks when using nested invocations.

An object adapter's thread pool supports all of the properties described in Configuring Thread Pools. For configuration purposes, the name of an adapter's thread pool is *`adapter`*`.ThreadPool`, where *`adapter`* is the name of the adapter.

An adapter creates its own thread pool when at least one of the following properties has a value greater than zero:

- *`adapter`*`.ThreadPool.Size`
- *`adapter`*`.ThreadPool.SizeMax`

These properties have the same semantics as those described earlier except they both have a default value of zero, meaning that an adapter uses the communicator's thread pools by default.

As an example, the properties shown below configure a thread pool for the object adapter named `PrinterAdapter`:

```
PrinterAdapter.ThreadPool.Size=3
PrinterAdapter.ThreadPool.SizeMax=15
PrinterAdapter.ThreadPool.SizeWarn=14
```

See Also

- Thread Pools
- Object Adapters
- Nested Invocations
- Object Adapter Properties

Thread Pool Design Considerations

Improper configuration of a thread pool can have a serious impact on the performance of your application. This page discusses some issues that you should consider when designing and configuring your applications.

On this page:

Single-Threaded Pool

There are several implications of using a thread pool with a maximum size of one thread:

- **Only one message can be dispatched at a time.**

  This can be convenient because it lets you avoid (or postpone) dealing with thread-safety issues in your application. However, it also eliminates the possibility of dispatching requests concurrently, which can be a bottleneck for applications running on multi-CPU systems or that perform blocking operations. Another option is to enable serialization in a multi-threaded pool.

- **Only one AMI reply can be processed at a time.**

  An application must increase the size of the client thread pool in order to process multiple AMI callbacks in parallel.

- **Nested twoway invocations are limited.**

  At most one level of nested twoway invocations is possible.

It is important to remember that a communicator's client and server thread pools have a default maximum size of one thread, therefore these limitations also apply to any object adapter that shares the communicator's thread pools.

Multi-Threaded Pool

Configuring a thread pool to support multiple threads implies that the application is prepared for the Ice run time to dispatch operation invocations or AMI callbacks concurrently. Although greater effort is required to design a thread-safe application, you are rewarded with the ability to improve the application's scalability and throughput.

Choosing an appropriate maximum size for a thread pool requires careful analysis of your application. For example, in compute-bound applications it is best to limit the number of threads to the number of physical processor cores or threads on the host machine; adding any more threads only increases context switches and reduces performance. Increasing the size of the pool beyond the number of cores can improve responsiveness when threads can become blocked while waiting for the operating system to complete a task, such as a network or file operation. On the other hand, a thread pool configured with too many threads can have the opposite effect and negatively impact performance. Testing your application in a realistic environment is the recommended way of determining the optimum size for a thread pool.

If your application uses nested invocations, it is very important that you evaluate whether it is possible for thread starvation to cause a deadlock. Increasing the size of a thread pool can lessen the chance of a deadlock, but other design solutions are usually preferred.

Serializing Requests in a Multi-Threaded Pool

When using a multi-threaded pool, the nondeterministic nature of thread scheduling means that requests from the same connection may not be dispatched in the order they were received. Some applications cannot tolerate this behavior, such as a transaction processing server that must guarantee that requests are executed in order. There are two ways of satisfying this requirement:

1. Use a single-threaded pool.
2. Configure a multi-threaded pool to serialize requests using its `Serialize` property.

At first glance these two options may seem equivalent, but there is a significant difference: a single-threaded pool can only dispatch one request at a time and therefore serializes requests from *all* connections, whereas a multi-threaded pool configured for serialization can dispatch requests from different connections concurrently while serializing requests from the same connection.

> For requests dispatched using asynchronous method dispatch (AMD), `Serialize` only serializes the dispatching, not the requests themselves. Ice will dispatch the next request once its dispatch its complete–it does not wait for the first request to provide a response or exception.

You can obtain a comparable behavior from a multi-threaded pool without enabling serialization, but only if you design the clients so that they do not send requests from multiple threads, do not send requests over more than one connection, and only use synchronous twoway invocations. In general, however, it is better to avoid such tight coupling between the implementations of the client and server.

Enabling serialization can improve responsiveness and performance compared to a single-threaded pool, but there is an associated cost. The extra synchronization that the pool must perform to serialize requests can add significant overhead and result in higher latency and reduced throughput.

As you can see, thread pool serialization is not a feature that you should enable without analyzing whether the benefits are worthwhile. For example, it might be an inappropriate choice for a server with long-running operations when the client needs the ability to have several operations in progress simultaneously. If serialization was enabled in this situation, the client would be forced to work around it by opening several connections to the server, which again tightly couples the client and server implementations. If the server must keep track of the order of client requests, a better solution would be to use serialization in conjunction with AMD to queue the incoming requests for execution by other threads.

See Also

- Thread Pools
- Concurrent Proxy Invocations
- Nested Invocations
- Thread Safety
- Connection Establishment

Concurrent Proxy Invocations

Proxy objects are fully thread safe, meaning a client can invoke on the same proxy object from multiple threads concurrently without the need for additional synchronization. However, Ice makes few guarantees about the order in which it sends proxy invocations over a connection.

To understand the ordering issue, it's important to first understand some fundamental proxy concepts:

- At any point in time, a proxy may or may not be associated with a connection.
- A new proxy initially has no connection, and Ice does not attempt to associate it with a connection until its first invocation.
- If Ice needs to establish a new connection for a proxy, Ice queues all invocations on that proxy until the connection succeeds.
- After a proxy is associated with a connection, the proxy may or may not cache that connection for subsequent invocations.
- Proxies share connections by default, but an application can force proxies to use separate connections.

Ice guarantees that ordering will be maintained for invocations on the same proxy object, but only if that proxy caches its connection. This guarantee also holds for invocations on two or more proxies that programmatically compare as equal, meaning the proxies have the same identity and other configuration settings, and furthermore these proxies must cache connections as well.

Ice does not guarantee the order of invocations for any other situation.

> The order in which the Ice run time in a client sends invocations over a connection does not necessarily determine the order in which they will be executed in the server. See Thread Pool Design Considerations for information about ordering guarantees in servers.

See Also

- Proxies
- Connection Establishment
- Thread Pool Design Considerations

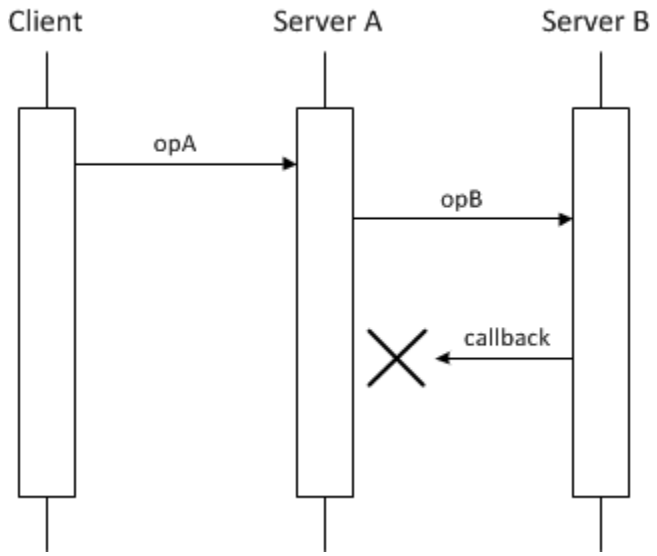
Nested Invocations

A *nested invocation* is one that is made within the context of another Ice operation. For instance, the implementation of an operation in a servant might need to make a nested invocation on some other object, or an AMI callback object might invoke an operation in the course of processing a reply to an asynchronous request. It is also possible for one of these invocations to result in a nested callback to the originating process. The maximum depth of such invocations is determined by the size of the thread pools used by the communicating parties.

On this page:

Deadlocks with Nested Invocations

Applications that use nested invocations must be carefully designed to avoid the potential for deadlock, which can easily occur when invocations take a circular path. For example, this illustration presents a deadlock scenario when using the default thread pool configuration:



Nested invocation deadlock.

In this diagram, the implementation of `opA` makes a nested twoway invocation of `opB`, but the implementation of `opB` causes a deadlock when it tries to make a nested callback. As mentioned in Thread Pools, the communicator's thread pools have a maximum size of one thread unless explicitly configured otherwise. In Server A, the only thread in the server thread pool is busy waiting for its invocation of `opB` to complete, and therefore no threads remain to handle the callback from Server B. The client is now blocked because Server A is blocked, and they remain blocked indefinitely unless timeouts are used.

There are several ways to avoid a deadlock in this scenario:

- **Increase the maximum size of the server thread pool in Server A.**

  Configuring the server thread pool in Server A to support more than one thread allows the nested callback to proceed. This is the simplest solution, but it requires that you know in advance how deeply nested the invocations may occur, or that you set the maximum size to a sufficiently large value that exhausting the pool becomes unlikely. For example, setting the maximum size to two avoids a deadlock when a single client is involved, but a deadlock could easily occur again if multiple clients invoke `opA` simultaneou sly. Furthermore, setting the maximum size too large can cause its own set of problems.

- **Use a oneway invocation.**

  If Server A called `opB` using a oneway invocation, it would no longer need to wait for a response and therefore `opA` could complete, making a thread available to handle the callback from Server B. However, we have made a significant change in the semantics of o `pA` because now there is no guarantee that `opB` has completed before `opA` returns, and it is still possible for the oneway invocation of `opB` to block.

- **Create another object adapter for the callbacks.**

  No deadlock occurs if the callback from Server B is directed to a different object adapter that is configured with its own thread pool.

- **Implement `opA` using asynchronous dispatch and invocation.**

  By declaring `opA` as an AMD operation and invoking `opB` using AMI, Server A can avoid blocking the thread pool's thread while it waits for `opB` to complete. This technique, known as *asynchronous request chaining*, is used extensively in Ice services such as IceGrid and Glacier2 to eliminate the possibility of deadlocks.

As another example, consider a client that makes a nested invocation from an AMI callback object using the default thread pool configuration. The (one and only) thread in the client thread pool receives the reply to the asynchronous request and invokes its callback object. If the callback object in turn makes a nested twoway invocation, a deadlock occurs because no more threads are available in the client thread pool to process the reply to the nested invocation. The solutions are similar to some of those presented in the above illustration: increase the maximum size of the client thread pool, use a oneway invocation, or call the nested invocation using AMI.

Analyzing an Application for Nested Invocations

A number of factors must be considered when evaluating whether an application is properly designed and configured for nested invocations:

- The thread pool configurations in use by all communicating parties have a significant impact on an application's ability to use nested invocations. While analyzing the path of circular invocations, you must pay careful attention to the threads involved to determine whether sufficient threads are available to avoid deadlock. This includes not just the threads that dispatch requests, but also the threads that make the requests and process the replies. Enabling the `Ice.Trace.ThreadPool` property can give you a better understanding of the thread pool behavior in your application.
- Bidirectional connections are another complication, since you must be aware of which threads are used on either end of the connection.
- Finally, the synchronization activities of the communicating parties must also be scrutinized. For example, a deadlock is much more likely when a lock is held while making an invocation.

As you can imagine, tracing the call flow of a distributed application to ensure there is no possibility of deadlock can quickly become a complex and tedious process. In general, it is best to avoid circular invocations if at all possible.

See Also

- Thread Pools
- Object Adapter Thread Pools
- Thread Pool Design Considerations
- Oneway Invocations

Thread Safety

The Ice run time itself is fully thread safe, meaning multiple application threads can safely call methods on objects such as communicators, object adapters, and proxies without synchronization problems. As a developer, you must also be concerned with thread safety because the Ice run time can dispatch multiple invocations concurrently in a server. In fact, it is possible for multiple requests to proceed in parallel within the same servant and within the same operation on that servant. It follows that, if the operation implementation manipulates non-stack storage (such as member variables of the servant or global or static data), you must interlock access to this data to avoid data corruption.

The need for thread safety in an application depends on its configuration. Using the default thread pool configuration typically makes synchronization unnecessary because at most one operation can be dispatched at a time. Thread safety becomes an issue once you increase the maximum size of a thread pool.

Ice uses the native synchronization and threading primitives of each platform. For C++ users, Ice provides a collection of convenient and portable wrapper classes for use by Ice applications.

On this page:

Threading Issues with Marshaling

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference:

- Ice is marshaling this data, outside any synchronization under the control of the application
- At the same time, another request updates the same very same data

Ice provides an elegant solution for this problem with the marshaled-result metadata, available for the following language mappings:

- C++11
- C#
- Java
- Python

Thread Creation and Destruction Hooks

On occasion, it is necessary to intercept the creation and destruction of threads created by the Ice run time, for example, to interoperate with libraries that require applications to make thread-specific initialization and finalization calls (such as COM's `CoInitializeEx` and `CoUninitialize`). Ice provides callbacks to inform an application when each run-time thread is created and destroyed.

The callback or callbacks are registered through the `InitializationData` parameter passed to `initialize`:

C++11 C++98 C# Java Java Compat Python

```
struct InitializationData
{
    // ...
    std::function<void()> threadStart;
    std::function<void()> threadStop;
};
```

```
class ThreadNotification : public IceUtil::Shared
{
public:
    virtual void start() = 0;
    virtual void stop() = 0;
};
typedef IceUtil::Handle<ThreadNotification> ThreadNotificationPtr;


struct InitializationData
{
    // ...
    ThreadNotificationPtr threadHook;
};
```

```
public class InitializationData
{
    // ...
    public System.Action threadStart;
    public System.Action threadStop;
}
```

```
public class InitializationData
{
    // ...
    public Runnable threadStart;
    public Runnable threadStop;
}
```

```
public interface ThreadNotification
{
    void start();
    void stop();
}


public class InitializationData
{
    // ...
    ThreadNotification threadHook;
}
```

```
initData = Ice.InitializationData()
initData.threadStart = lambda: # handle thread start...
initData.threadStop = lambda: # handle thread stop...
```

To receive notification of thread creation and destruction, you must implement and register these callbacks. They will be called by the Ice run time by each thread as soon as it is created, and just before it exits.

For example, you could define callbacks and register them with the Ice run time as follows:

C++11 C++98

```
int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.start = [] { cout << "start: id = " <<
std::this_thread::get_id() << endl; };
    initData.stop = [] { cout << "stop: id = " <<
std::this_thread::get_id() << endl; };
    Ice::CommunicatorHolder ich(argc, argv, initData);

    // ...
}
```

```
class MyHook : public Ice::ThreadNotification
{
public:
    void start()
    {
        cout << "start: id = " << ThreadControl().id() << endl;
    }
    void stop()
    {
        cout << "stop: id = " << ThreadControl().id() << endl;
    }
};

int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.threadHook = new MyHook;
    Ice::CommunicatorHolder ich(argc, argv, initData);

    // ...
}
```

Installing Thread Hooks with a Plug-in

The thread hook facility described above requires that you modify a program's source code in order to receive callbacks when threads in the Ice run time are created and destroyed. It is also possible to install thread hooks using the Ice plug-in facility, which is useful for adding thread hooks to an existing program that you cannot (or prefer not to) modify.

Ice provides a base class named `ThreadHookPlugin` for C++, Java, and C# that supplies the necessary functionality:

C++11 C++98 C# Java Java Compat

```
namespace Ice
{
    class ThreadHookPlugin : public Ice::Plugin
    {
    public:

        ThreadHookPlugin(const std::shared_ptr<Communicator>&
communicator, std::function<void()>, std::function<void()>);

        virtual void initialize();
        virtual void destroy();
    };
}
```

```
namespace Ice
{
    class ThreadHookPlugin : public Ice::Plugin
    {
    public:

        ThreadHookPlugin(const CommunicatorPtr& communicator,
const ThreadNotificationPtr&);

        virtual void initialize();
        virtual void destroy();
    };
}
```

```
namespace Ice
{
    public class ThreadHookPlugin : Plugin
    {
        public ThreadHookPlugin(Communicator communicator, System.Action
threadStart, System.Action threadStop) { ... }

        public void initialize() {}
        public void destroy() {}
    }
}
```

```
package com.zeroc.Ice;
public class ThreadHookPlugin implements Plugin
{
    public ThreadHookPlugin(Communicator communicator, Runnable
threadStart, Runnable threadStop) { ... }

    @Override
    public void initialize() {}

    @Override
    public void destroy() {}
}
```

```
package Ice;
public class ThreadHookPlugin implements Plugin
{
    public ThreadHookPlugin(Communicator communicator,
ThreadNotificationHook threadHook) { ... }

    @Override
    public void initialize() {}

    @Override
    public void destroy() {}
}
```

The `ThreadHookPlugin` constructor installs the given thread callbacks into the specified communicator. The `initialize` and `destroy` methods are empty, but you can subclass `ThreadHookPlugin` and override these methods if necessary.

In order to create a thread hook plug-in, you must do the following:

- Define and export a factory class (for Java and C#) or factory function (for C++) that returns an instance of `ThreadHookPlugin`, as described in the plug-in API.
- Implement the callback(s) that you will pass to the `ThreadHookPlugin` constructor.
- Package your code into a format that is suitable for dynamic loading, such as a shared library or DLL for C++ or an assembly for C#.

See the Plug-in Facility for more details on how to package and register your plug-in.

See Also

- Communicator Initialization
- Plug-in Facility
- Plug-in API
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

Dispatching Requests to User Threads

By default, Ice uses a thread from one of its thread pools to dispatch requests and to execute callbacks registered to run upon completion of asynchronous invocations.

For regular calls, Ice uses threads from its server thread pool (or object adapter thread pool, if one is configured) for dispatching requests, and it uses threads from its client thread pool to execute AMI callbacks. For bidir calls, the roles are reversed: Ice uses threads from its client thread pool for dispatching requests and threads from the server thread pool (or object adapter thread pool) for AMI callbacks.

This simple behavior is suitable for most applications. There are however situations you may want to execute dispatches or AMI callbacks in a particular thread. For example, in a server, you might need to update a database that does not permit concurrent access from different threads or, in a client, you might need to update a user interface with the results of an invocation. (Many UI frameworks require all UI updates to be made by a specific thread.)

Ice allows you to register a *dispatcher* to control which thread(s) execute dispatches and AMI callbacks. The dispatcher API is specific to each language mapping.

On this page:

Dispatcher Interception Point

A dispatcher intercepts a call (request dispatch or AMI callback) and can select in which thread to execute this call. An Ice communicator gives each such call to the registered dispatcher before this call unmarshals parameters and before this call locates the target servant (for request dispatches). The call given to the dispatcher takes no parameters, returns nothing, and never throws any exception back to the dispatcher.

Registering the Dispatcher in InitializationData

You configure the dispatcher used by an Ice communicator by setting the data member `dispatcher` of its `InitializationData`:

C++11 C++98 C# Java Java CompatObjC Python

```
int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.properties = Ice::createProperties(argc, argv);
    initData.dispatcher = dispatcherFunction;

    Ice::CommunicatorHolder ich(argc, argv, initData);

    // ...
}
```

The `dispatcher` data member's type is `std::function<void(std::function<void()>, const std::shared_ptr<Ice::Connection>&)>`.

```
class MyDispatcher : public Ice::Dispatcher /*, ... */
{
    // ...
};

int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.properties = Ice::createProperties(argc, argv);
    initData.dispatcher = new MyDispatcher;
    Ice::CommunicatorHolder ich(argc, argv, initData);
    ...
}
```

The `Ice::Dispatcher` abstract base class has the following interface:

```
namespace Ice
{
    class Dispatcher : public virtual IceUtil::Shared
    {
    public:
        virtual void dispatch(const DispatcherCallPtr&,
const ConnectionPtr&) = 0;
    };

    typedef IceUtil::Handle<Dispatcher> DispatcherPtr;
}
```

The `DispatcherCall` instance encapsulates all the details of the call. It is another abstract base class with the following interface:

```
namespace Ice
{
    class DispatcherCall : public virtual IceUtil::Shared
    {
    public:
        virtual ~DispatcherCall() { }

        virtual void run() = 0;
    };

    typedef IceUtil::Handle<DispatcherCall> DispatcherCallPtr;
}
```

C

```
public class Server
{
    public static void Main(string[] args)
    {
        Ice.InitializationData initData = new Ice.InitializationData();
        initData.dispatcher = (System.Action call, Ice.Connection
connection) => { ... };
        using(Ice.Communicator communicator =
Ice.Util.initialize(ref args, initData))
        {
            // ...
        }
    }
}
```

The dispatcher is a delegate with type `System.Action<System.Action, Ice.Connection>`.

```
public class Server
{
    public static void main(String[] args)
    {
        com.zeroc.Ice.InitializationData initData = new
com.zeroc.Ice.InitializationData();
        initData.dispatcher = (runnable, connection) -> { ... };

        try(com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(args, initData))
        {
            // ...
        }
        catch(LocalException ex)
        {
            // ...
        }

        // ...
    }
}
```

The `dispatcher` is an object (typically a lambda) that implements the functional interface `java.util.function.BiConsumer<Runnable, com.zeroc.Ice.Connection>`.

```
public class MyDispatcher implements Ice.Dispatcher
{
    // ...
}

public class Server
{
    public static void main(String[] args)
    {
        Ice.InitializationData initData = new Ice.InitializationData();
        initData.dispatcher = new MyDispatcher();
        try(Ice.Communicator communicator = Ice.Util.initialize(args,
initData))
        {
           // ...
        }
        // ...
    }
}
```

The `dispatcher` is an object that implements the functional interface `Ice.Dispatcher`:

```
public interface Dispatcher
{
    void dispatch(Runnable runnable, Ice.Connection con);
}
```

```
int
main(int argc, char* argv[])
{
    objc_startCollectorThread();
    id<ICECommunicator> communicator = nil;
    @try
    {
        ICEInitializationData* initData = [ICEInitializationData
initializationData];
        initData.dispatcher =
            ^(id<ICEDispatcherCall> call, id<ICEConnection> con)
            {
                // ...
            };
        communicator = [ICEUtil createCommunicator:&argc argv:argv
initData:initData];
        // ...
    }
    @catch(ICELocalException* ex)
    {
        // ...
    }

    // ...
}
```

The type of the dispatcher callback must match the following block signature:

```
void(^)(id<ICEDispatcherCall> call, id<ICEConnection> connection)
```

The `ICEDispatcherCall` protocol defines how to execute the call:

```
@protocol ICEDispatcherCall <NSObject>
-(void) run;
@end
```

```
initData = Ice.InitializationData()
initData.dispatcher = lambda call, connection: ...

with Ice.Util.initialize(sys.argv, initData) as communicator:
    # ...
```

The dispatcher is a callable object.

Afterwards, the Ice communicator calls the configured dispatcher whenever it dispatches a request or executes an application-supplied callback upon completion of an asynchronous invocation. The first parameter given to the dispatcher corresponds to the call (dispatch or AMI

callback) that the dispatcher must execute. The second parameter is the connection associated with this call (if any). The connection parameter is null in the following situations:

- for collocated calls
- when the call failed before a connection could be associated with this call
- when the call is executed through a C# scheduler or Java executor (see Dispatcher with Future Results later on this page)

> A dispatcher must always execute the supplied call. Failure to dispatch a call will cause `Communicator::destroy` to block indefinitely. Furthermore, a dispatcher must not make blocking calls from the dispatch thread, such as synchronous invocations or calls to proxy methods that can potentially block, like `ice_getConnection`. Since these calls use the dispatcher for their own completion, you will get a deadlock if your dispatcher executes all calls on a single thread.
>
> If a dispatcher has resources that must be reclaimed (e.g., joining with a helper thread), it can safely do so after `Communicator::destroy` has completed.

Trivial Dispatcher

You can write a dispatcher that blocks and waits for completion of the supplied call, since the dispatcher is called by a thread in the server-side thread pool (for non-bidir request dispatches) or the client-side thread pool (for non-bidir AMI callbacks). For example:

C++11 C++98 C# Java Java Compat Objective-C Python

```
initData.dispatcher = [](std::function<void()> dispatchCall, const
std::shared_ptr<Ice::Connection>&)
                      {
                         dispatchCall(); // Does not throw, blocks until
the call completes.
                      };
```

```
class MyDispatcher : public Ice::Dispatcher
{
public:
    virtual void dispatch(const Ice::DispatcherCallPtr& d,
const Ice::ConnectionPtr&)
    {
        d->run(); // Does not throw, blocks until the call completes.
    }
};
```

```
initData.dispatcher = (System.Action call, Ice.Connection connection) =>

{
    call(); // Does not throw, blocks until the call completes.
}
```

```
initData.dispatcher = (runnable, connection) -> { runnable.run(); };
```

```
public class MyDispatcher implements Ice.Dispatcher
{
    public void dispatch(Runnable runnable, Ice.Connection connection)
    {
        // Does not throw, blocks until the call completes.
        runnable.run();
    }
}
```

```
void(^myDispatcher)(id<ICEDispatcherCall>, id<ICEConnection>) =
    ^(id<ICEDispatcherCall> call, id<ICEConnection> con)
    {
        // Does not throw, blocks until the call completes.
        [call run];
    };
```

```
initData.dispatcher = lambda call, connection: call()
```

This implementation ties up a thread in the thread pool for the duration of the call, and is not particularly useful: a communicator with no dispatcher provides the same behavior.

Dispatcher for Graphical Applications

The primary use-case for dispatchers is graphical applications where only one thread is allowed to call UI methods. With such an application, you can register a dispatcher that executes all calls in the UI thread. You can also use the UI thread to make asynchronous invocations, since Ice guarantees asynchronous invocations never block the calling thread.

Here are some examples:

C++11 C++98 C# Java ObjC

With Qt

```
//
// Define a custom event type to be used by the dispatcher
//
class DispatchEvent : public QEvent
{
public:

    DispatchEvent(std::function<void()> call) :
        QEvent(QEvent::Type(CUSTOM_EVENT_TYPE)),
        _call(call)
    {
    }

    void dispatch()
```

```
    {
        _call();
    }

private:

    std::function<void()> _call;
};

MainWindow::MainWindow()
{
    Ice::InitializationData initData;
    initData.properties = Ice::createProperties();
    //
    // The dispatcher implementation creates a new DispatchEvent and
adds it to event
    // queue, setting this object as the receiver of the event.
    //
    initData.dispatcher = [this](std::function<void()> dispatchCall,
const std::shared_ptr<Ice::Connection>&)
    {
        QApplication::postEvent(this, new DispatchEvent(dispatchCall));
    };
    _communicator = Ice::initialize(initData);
}

//
// Override QObject::event to handle our custom event type and delegate
// to the base class to handle other event types.
//
bool
MainWindow::event(QEvent* event)
{
    if(event->type() == CUSTOM_EVENT_TYPE)
    {
        auto dispatchEvent = static_cast<DispatchEvent*>(event);
        try
        {
            dispatchEvent->dispatch();
        }
        catch(const std::exception& ex)
        {
           ...
        }
        return true;
```

```
    }
    return QMainWindow::event(event);
}
```

With MFC

```
class MyDialog : public CDialog { ... };

class MyDispatcher : public Ice::Dispatcher
{
public:
    MyDispatcher(MyDialog* dialog) : _dialog(dialog)
    {
    }

    virtual void
    dispatch(const Ice::DispatcherCallPtr& call,
const Ice::ConnectionPtr&)
    {
        _dialog->PostMessage(WM_AMI_CALLBACK, 0,

reinterpret_cast<LPARAM>(new Ice::DispatcherCallPtr(call)));
    }

private:
    MyDialog* _dialog;
};
```

The `MyDispatcher` class simply stores the `CDialog` handle for the UI and calls `PostMessage`, passing the `DispatcherCall` instance. In turn, this causes the UI thread to receive an event and invoke the UI callback method that was registered to respond to `WM_AMI_CALLBAC K` events.

The implementation of the callback function calls `run`:

With MFC

```
LRESULT
MyDialog::OnAMICallback(WPARAM, LPARAM lParam)
{
    try
    {
        Ice::DispatcherCallPtr* call =
reinterpret_cast<Ice::DispatcherCallPtr*>(lParam);
        (*call)->run();
        delete call;
    }
    catch(const Ice::Exception& ex)
    {
        // ...
    }
    return 0;
}
```

The Ice run time calls `dispatch` once the asynchronous invocation is complete. In turn, this triggers the `OnAMICallback` which calls `run`. Because the operation has completed already, `run` does not block, so the UI remains responsive.

With WPF

```
public partial class MyWindow : Window
{
    private void Window_Loaded(object sender, EventArgs e)
    {
        Ice.InitializationData initData = new Ice.InitializationData();
        initData.dispatcher = (System.Action call, Ice.Connection
connection) =>
        {
            Dispatcher.BeginInvoke(DispatcherPriority.Normal, action);
        };
        using(Ice.Communicator communicator =
Ice.Util.initialize(initData))
        {
            // ...
        }
     }
}
```

The delegate calls `Dispatcher.BeginInvoke` on the `action` delegate. This causes WPF to queue the actual asynchronous invocation of `action` for later execution by the UI thread. Because the Ice run time does not call your delegate until an asynchronous invocation is complete, when the UI thread executes the corresponding call to the `EndInvoke` method, that call does not block and the UI remains responsive.

The net effect is that you can invoke an operation asynchronously from a UI callback method without the risk of blocking the UI thread. For example:

With WPF

```
public partial class MyWindow : Window
{
    private async void someOp_Click(object sender, RoutedEventArgs e)
    {
        MyIntfPrx p = ...;

        try
        {
            // Call remote operation asynchronously.
            await p.someOpAsync();
            // Update UI...
        }
        catch(System.Exception ex)
        {
            // Update UI...
        }
    }
}
```

We're using Ice's task-based API together with the `async` and `await` keywords to execute asynchronous tasks in a straightforward way. The return value of `someOpAsync` is a `Task` on which we use the `await` keyword to suspend processing until the call completes. Thanks to the dispatcher, processing eventually resumes in the UI thread and we can update the UI as needed. The `csharp/Ice/wpf` demo shows a fully-functional UI client that uses this technique.

With Swing

```
public class Client extends JFrame
{
    public static void main(final String[] args)
    {
        SwingUtilities.invokeLater(
            () -> {
                      try
                      {
                          new Client(args);
                      }
                      catch(com.zeroc.Ice.LocalException e)
                      {
                          JOptionPane.showMessageDialog(
                             null, e.toString(),
                             "Initialization failed",
                             JOptionPane.ERROR_MESSAGE);
                      }
                  });
    }

    Client(String[] args)
    {
        com.zeroc.Ice.InitializationData initData = new
com.zeroc.Ice.InitializationData();
        initData.dispatcher = (runnable, connection) -> {
SwingUtilities.invokeLater(runnable); };
        try(com.zeroc.Ice.Communicator communicator =
com.zeroc.Ice.Util.initialize(args, initData))
        {
            ...
        }
    }
}
```

The dispatcher simply delays the call to `run` by calling `invokeLater`, passing it the `Runnable` that is provided by the Ice run time. This causes the Swing UI thread to make the call to `run`. Because the Ice run time does not call the dispatcher until the asynchronous invocation is complete, that call to `run` does not block and the UI remains responsive.

The `java/Ice/swing` demo shows a fully-functional UI client that uses this technique.

With Cocoa

```
-(void)viewDidLoad
{
    ICEInitializationData* initData = [ICEInitializationData
initializationData];
    initData.dispatcher =
        ^(id<ICEDispatcherCall> call, id<ICEConnection> con)
        {
            dispatch_sync(dispatch_get_main_queue(), ^ { [call run]; });
        };

    communicator = [[ICEUtil createCommunicator:initData] retain];

    // ....
}
```

The dispatcher callback calls `dispatch_sync` on the main queue. This queues the actual call for later execution by the main thread. Because the Ice run time does not call the dispatcher callback until an asynchronous operation invocation is complete, when the UI thread executes the corresponding call, that call does not block and the UI remains responsive.

The net effect is that you can invoke an operation asynchronously from a UI callback method without the risk of blocking the UI thread. For example:

With Cocoa

```
-(void)someOp:(id)sender
{
    id<MyIntfPrx> p = ...;
    [p begin_someOp:^{ [self response]; }
          exception:^(ICEException* ex) { [self exception:ex]; }];
}

-(void) response
{
    // Update UI...
}

-(void) exception:(ICEException* ex)
{
    // Update UI...
}
```

The `objective-c/Ice/iOS/hello` demo shows a fully-functional UI client that uses this technique.

Dispatcher with Future Results

With C#, Java and Python, asynchronous method invocations return a "future" object on which you can register callbacks to execute upon completion. This future object is a Task in C#, a CompletableFuture in Java and an Ice.InvocationFuture in Python. (We are ignoring here the deprecated AMI APIs for C# and Python that don't return future objects.)

In C# and Java, the thread that executes the callback you supply is determined by the .NET and Java specifications, respectively, so unless you are careful, you typically won't use the configured dispatcher for these callbacks.

It is also possible that the call has already completed by the time you register the callback with the future object, so depending on how you register the callback, the callback could execute synchronously from the calling thread.

If you want to always use the configured dispatcher for these callbacks, regardless of whether or not the call already completed when you register the callback, you need to:

- in C#, register your callback with `ContinueWith` using a custom scheduler provided by *`proxy`*`.ice_scheduler()`. *`proxy`* represents the proxy you used for the asynchronous invocation. You should also not specify the `ExecuteSynchronously` task continuation option or you should use the `RunContinuationsAsynchronously` option to ensure callbacks are always executed asynchronously even if the task completed when `ContinueWith` is called.
- in Java, call an `Async` method on the `CompletableFuture` result (such as `whenCompleteAsync`), and pass *`proxy`*`.ice_executor()` as the second parameter. *`proxy`* represents the proxy you used for the asynchronous invocation.
- in Python, register your callback with `add_done_callback_async` instead of `add_done_callback.`

See Also

- Asynchronous Method Invocation (AMI) in C++11
- Asynchronous Method Invocation (AMI) in C++98
- Asynchronous Method Invocation (AMI) in C-Sharp
- Asynchronous Method Invocation (AMI) in Java
- Asynchronous Method Invocation (AMI) in Java Compat
- Asynchronous Method Invocation (AMI) in Objective-C
- Asynchronous Method Invocation (AMI) in Python

Blocking API Calls

This page lists the Ice APIs that can potentially block the calling thread. Graphical applications should normally avoid calling these APIs from the "event loop" thread.

> The JavaScript mapping does not have any blocking APIs. The operations listed below are implemented as asynchronous operations.

Synchronous Invocations

All synchronous remote invocations can block, including the built-in proxy operations `ice_ping`, `ice_isA`, `ice_id` and `ice_ids`. Since `checkedCast` internally calls `ice_isA`, it too can block.

Asynchronous Invocations

For the C++98, C#, Python and Java Compat language mappings that still support the older `begin`/`end` API for asynchronous remote invocations, the `begin` method never blocks but the `end` method can block. Furthermore, the `AsyncResult` object returned by `begin` defines two methods that can potentially block: `waitForCompleted` and `waitForSent`.

In the C++11, C#, Python and Java language mappings, the *op*`Async` method never blocks, but the future or task object it returns provides an accessor that can block indefinitely until a result or exception is set.

Local APIs

Ice also provides a number of local APIs that can block, either because they wait indefinitely for a condition to become true, or because their implementations may make remote invocations:

- Communicator
  - `createAdmin`
  - `createObjectAdapter`
  - `createObjectAdapterWithEndpoints`
  - `createObjectAdapterWithRouter`
  - `destroy`
  - `end_flushBatchRequests` (C++98, C#, Python, Java Compat)
  - `flushBatchRequests`
  - `getAdmin`
  - `shutdown`
  - `waitForShutdown`
- `Connection`
  - `close`
  - `end_flushBatchRequests` (C++98, C#, Python, Java Compat)
  - `flushBatchRequests`
- `ObjectPrx`
  - `ice_getConnection`
  - `end_ice_getConnection` (C++98, C#, Python, Java Compat)
- `ObjectAdapter`
  - `activate`
  - `deactivate`
  - `destroy`
  - `waitForDeactivate`
  - `waitForHold`

> The Java and Java Compat mappings allow you to interrupt threads blocked in these APIs.

Connection Management

The Ice run time establishes connections automatically and transparently as a side effect of using proxies. There are well-defined rules that determine when a new connection is established. If necessary, you can influence connection management activities.

Connection management becomes increasingly important as network environments grow more complex. In particular, if you need to make callbacks from a server to a client through a firewall, you must use a bidirectional connection. In most cases, you can use a Glacier2 router t o automatically take advantage of bidirectional connections. However, the Ice run time also provides direct access to connections, allowing you to explicitly control establishment and closure of both unidirectional and bidirectional connections.

Most Ice applications benefit from active connection management and transparent connection establishment and thus need not concern themselves with the details of connections. Not all Ice applications can be so fortunate, and for those applications Ice provides convenient ac cess to connections that enables developers to address the realities of today's deployment environments.

> The discussion that follows assumes that you are familiar with proxies and endpoints.

Topics

Connection Establishment

Connections are established as a side effect of using proxies. The first invocation on a proxy causes the Ice run time to search for an existing connection to one of the proxy's endpoints; only if no suitable connection exists does the Ice run time establish a new connection to one of the proxy's endpoints.

This page describes how and when Ice establishes a new connection.

On this page:

Endpoint Selection for New Connections

A proxy performs a number of operations on its endpoints before it asks the Ice run time to supply a connection. These operations produce a list of zero or more endpoints that satisfy the proxy's configuration. If the resulting list is empty, the application receives `NoEndpointException` to indicate that no suitable endpoints could be found. For example, this situation can arise when a twoway proxy contains only a UDP endpoint; the UDP endpoint is eliminated from consideration because it cannot be used for twoway invocations.

The proxy performs the following steps to derive its endpoint list:

1. Remove the endpoints of unknown transports. For instance, SSL endpoints are removed if the SSL plug-in is not installed.
2. Remove endpoints that are not suitable for the proxy's invocation mode. For example, datagram endpoints are removed for twoway, oneway and batch oneway proxies. Similarly, non-datagram endpoints are removed for datagram and batch datagram proxies.
3. Perform DNS queries to convert host names into IP addresses, if necessary. For a multi-homed host name, the proxy adds a new endpoint for each address returned by the DNS query.
4. Sort the endpoints according to the configured selection type, which is established using the `ice_endpointSelection` proxy method. The default value is `Random`, meaning the endpoints are randomly shuffled. Alternatively, the value `Ordered` maintains the existing order of the endpoints.
5. Satisfy the proxy's security requirements:
   - If `Ice.Override.Secure` is defined, remove all non-secure endpoints.
   - Otherwise, if the proxy is configured to prefer secure endpoints (e.g., by calling the `ice_preferSecure` proxy method), move all secure endpoints to the beginning of the list. Note that this setting still allows non-secure endpoints to be included.
   - Otherwise, move all non-secure endpoints to the beginning of the list.

If connection caching is enabled and the Ice run time already has a compatible connection, it reuses the cached connection. Otherwise, the run time attempts to connect to each endpoint in the list until it succeeds or exhausts the list; the order in which endpoints are selected for connection attempts depends on the endpoint selection policy. This policy can be set using a default property (`Ice.Default.EndpointSelection`), using a proxy property (*`name`*`.EndpointSelection`), and using the `ice_endpointSelection` proxy method.

Error Semantics for Failed Connections

If a failure occurs during a connection attempt, the Ice run time tries to connect to all of the proxy's remaining endpoints until either a connection is successfully established or all attempts have failed. At that point, the Ice run time may attempt automatic retries depending on the value of the `Ice.RetryIntervals` configuration property. The default value of this property is `0`, which causes the Ice run time to try connecting to all of the endpoints one more time.

> **Tip**
> Define the property `Ice.Trace.Retry=2` to monitor these attempts.

If no connection can be established on this second attempt, the Ice run time raises an exception that indicates the reason for the final failed attempt (typically `ConnectFailedException`). Similarly, if a connection was lost during a request and could not be reestablished (assuming the request can be retried), the Ice run time raises an exception that indicates the reason for the final failed attempt.

Reusing an Existing Connection

When establishing a connection for a proxy, the Ice run time reuses an existing connection when all the following conditions are met:

- The proxy has connection caching enabled.
- The remote endpoint matches one of the proxy's endpoints.
- The connection was established by the communicator that created the proxy.
- The connection matches the proxy's configuration. Connection timeout values play an important role here, as an existing connection is only reused if its connection timeout value (i.e., the connection timeout used when the connection was established) matches the endpoint timeout in the new proxy. Similarly, a proxy configured with a connection ID only reuses a connection if it was established by a proxy with the same connection ID.

When a proxy has connection caching disabled, the Ice run time does not prefer an endpoint with an already established connection over other endpoints. It can select an endpoint without an established connection and create a new connection; or it can select an endpoint with an established connection and reuse that connection.

Applications must exercise caution when using proxies containing multiple endpoints, especially endpoints using different transports. For example, suppose a proxy has multiple endpoints, such as one each for tcp and ssl. When establishing a connection for this proxy, the Ice run time will open a new connection only if it cannot reuse an existing connection to any of the endpoints (assuming connection caching is enabled). Furthermore, the proxy in its default (that is, non-secure) configuration gives higher priority to non-secure endpoints. If you want to ensure that a particular transport is used by a proxy, you must create the appropriate proxy, for example by calling the proxy method `ice_secure`.

Protocol Compression and Connection Reuse

The Ice run time does not consider compression settings when searching for existing connections to reuse; proxies whose compression settings differ can share the same connection (assuming all other selection criteria are satisfied).

Influencing Connection Reuse

The default behavior of the Ice run time, which reuses connections whenever possible, is appropriate for many applications because it conserves resources and typically has little or no impact on performance. However, when a server implementation attaches semantics to a connection, the client often must be designed to cooperate, despite the tighter coupling it causes. For example, a server might use a serialized thread pool to preserve the order of requests received over each connection. If the client wants to execute several requests simultaneously, it must be able to force the Ice run time to establish new connections at will.

For those situations that require more control over connection reuse, the Ice run time allows you to form arbitrary groups of proxies that share a connection by configuring them with the same connection identifier. The proxy method `ice_connectionId` returns a new proxy configured with the given connection ID. Once configured, the Ice run time ensures that the proxy only reuses a connection that was established by a proxy with the same connection ID (assuming all other criteria for connection reuse are also satisfied). A new connection is created if none with a matching ID is found, which means each proxy could conceivably have its own connection if each were assigned a unique connection ID.

As an example, consider the following code fragment:

C++11 C++98

```
auto prx = communicator->stringToProxy("ident:tcp -p 10000");
auto = g1 = prx->ice_connectionId("group1");
auto g2 = prx->ice_connectionId("group2");
prx->ice_ping(); // Opens a new connection
g1->ice_ping(); // Opens a new connection
g2->ice_ping(); // Opens a new connection
auto i1 = Ice::checkedCast<MyInterfacePrx>(g1);
i1->ice_ping(); // Reuses g1's connection
auto i2 =
Ice::checkedCast<MyInterfacePrx>(prx->ice_connectionId("group2"));
i2->ice_ping(); // Reuses g2's connection
```

```
Ice::ObjectPrx prx = comm->stringToProxy("ident:tcp -p 10000");
Ice::ObjectPrx g1 = prx->ice_connectionId("group1");
Ice::ObjectPrx g2 = prx->ice_connectionId("group2");
prx->ice_ping(); // Opens a new connection
g1->ice_ping(); // Opens a new connection
g2->ice_ping(); // Opens a new connection
MyInterfacePrx i1 = MyInterfacePrx::checkedCast(g1);
i1->ice_ping(); // Reuses g1's connection
MyInterfacePrx i2 =
MyInterfacePrx::checkedCast(prx->ice_connectionId("group2"));
i2->ice_ping(); // Reuses g2's connection
```

A total of three connections are established by this example:

1. The proxy `prx` establishes a new connection. This proxy has the default connection ID (an empty string).
2. The proxy `g1` establishes a new connection because the only existing connection, the one established by `prx`, has a different connection ID.
3. Similarly, the proxy `g2` establishes a new connection because none of the existing connections have a matching connection ID.

The proxy `i1` inherits its connection ID from `g1`, and therefore shares the connection for `group1`; `i2` explicitly configured its connection ID and shares the `group2` connection with proxy `g2`.

Connection Caching

When we refer to a proxy's connection, we actually mean the connection that the proxy is *currently* using. This connection can change over time, such that a proxy might use several connections during its lifetime. For example, an idle connection may be closed automatically and then transparently replaced by a new connection when activity resumes.

After establishing a connection in response to proxy activities, the Ice run time adds the connection to an internal pool for subsequent reuse by other proxies. The Ice run time manages the lifetime of the connection and eventually closes it. The connection is not affected by the life cycle of the proxies that use it, except that the lack of activity may prompt the Ice run time to close the connection after a while.

Once a proxy has been associated with a connection, the proxy's default behavior is to continue using that connection for all subsequent requests. In effect, the proxy caches the connection and attempts to use it for as long as possible in order to minimize the overhead of creating new connections. If the connection is later closed and the proxy is used again, the proxy repeats the connection-establishment procedure described earlier.

There are situations in which this default caching behavior is undesirable, such as when a client has a proxy with multiple endpoints and wishes to balance the load among the servers at those endpoints. The client can disable connection caching by passing an argument of `false` to the proxy factory method `ice_connectionCached`. The new proxy returned by this method repeats the connection-establishment

procedure before each request, thereby achieving request load balancing at the expense of potentially higher latency. This type of load balancing is performed solely by the client using whatever endpoints are contained in the proxy. More sophisticated forms of load balancing are also possible, such as when using IceGrid.

Enabling or disabling connection caching on a proxy has two separate effects:

- when caching is enabled, the proxy remembers ("caches") a connection until the connection is closed, while when caching is disabled, the proxy does not remember the connection it previously used.
- when caching is enabled, Ice provides to this proxy an already established connection if possible; when caching is disabled, Ice does not prefer endpoints with established connections over other endpoints when providing a connection to this proxy.

Timeouts and Connection Establishment

The default timeout for all connections is 60 seconds, as determined by the `Ice.Default.Timeout` property. This connection timeout value applies to all network operations. If a connection cannot be established within the allotted time, Ice raises `ConnectTimeoutException`.

You can set a connection timeout on a proxy using the `ice_timeout` proxy method. To use the same connection timeout for all proxies, you can define the `Ice.Override.Timeout` property; in this case, Ice ignores any connection timeout established using the `ice_timeout` proxy method or the `Ice.Default.Timeout` property. Finally, if you want to specify a separate timeout value that affects only connection establishment and takes precedence over a proxy's configured timeout value, you can define the `Ice.Override.ConnectTimeout` property.

Connection timeout values affect connection reuse. For example, if the endpoint in proxy A is identical to the endpoint in proxy B except their connection timeout values differ, the proxies cannot share the same connection.

The timeout in effect when a connection is established is bound to that connection and cannot be changed. If a network operation times out, all outstanding requests on that connection receive a `TimeoutException` and the connection is closed forcefully. The Ice run time automatically retries these requests on a new connection, assuming that automatic retries are enabled and would not violate at-most-once semantics.

> Invocation timeouts are a separate feature and do not affect connection reuse.

Source Address for New Connections

You can force Ice to use a specific source address for TCP/IP connections. This can be useful in specific uses cases, such as to overcome the limited number of ephemeral ports. There are two ways to do this:

- Define the `Ice.Default.SourceAddress` property to establish a default source address for all outgoing connections created by a communicator
- Include a `--sourceAddress` option in proxy endpoints, which overrides any setting for `Ice.Default.SourceAddress`

The value in each case must be an IP address.

See Also

- Proxy Methods
- Proxy Endpoints
- The Ice Threading Model
- Automatic Retries
- Connection Timeouts
- Active Connection Management
- IceGrid
- Ice.Default.*
- Proxy Properties
- Miscellaneous Ice.* Properties

Active Connection Management

Active Connection Management (ACM) helps to improve scalability and conserve application resources by closing idle connections. This feature is enabled by default.

On this page:

Configuring Active Connection Management

There are three components to an ACM configuration:

- Close behavior
  Determines the situations in which Ice closes a connection
- Heartbeat behavior
  Determines the situations in which Ice sends "heartbeat" messages to keep a connection alive
- Timeout
  Defines the time interval in which the Close and Heartbeat actions occur

You can configure these components using ACM properties. As explained below, there are four categories of ACM properties defined by the types of connections that the properties affect: global, client, server, and object adapter.

- `Ice.ACM.Close`
  `Ice.ACM.Heartbeat`
  `Ice.ACM.Timeout`
  These properties establish the global default settings for a communicator and affect both client (outgoing) and server (incoming) connections.
- `Ice.ACM.Client.Close`
  `Ice.ACM.Client.Heartbeat`
  `Ice.ACM.Client.Timeout`
  These properties override the default properties above for client (outgoing) connections.
- `Ice.ACM.Server.Close`
  `Ice.ACM.Server.Heartbeat`
  `Ice.ACM.Server.Timeout`
  These properties override the default properties above for all server (incoming) connections across all object adapters.
- *`adapter`*`.ACM.Close`
  `adapter``.ACM.Heartbeat`
  `adapter``.ACM.Timeout`
  These properties override the `Ice.ACM.Server` properties above for connections to a particular object adapter.

You can also configure the ACM components programmatically by calling `setACM` on a specific connection object. These settings override all ACM property configurations for that connection.

ACM Timeout Semantics

That the Close and Heartbeat behaviors share a single Timeout setting may seem surprising at first glance. For instance, it's possible for a program to configure certain combinations of Close and Heartbeat settings such that the heartbeats prevent a connection from ever becoming eligible for closure. In general, however, one peer will configure Close and Timeout settings while the other peer will configure Heartbeat and Timeout settings, in a coordinated effort to meet the application's requirements for connection management.

For each connection configured for ACM, Ice checks its status approximately every (Timeout / 2) seconds. If heartbeats are configured for the connection, Ice also sends a heartbeat during every status check. If two peers use the same Timeout value, this strategy ensures that heartbeat messages are sent well before the connection would be considered idle by the other peer.

Setting the timeout value to 0 disables ACM: connections won't be closed by ACM and heartbeats won't be sent.

Creating a Connection Management Policy

The ACM settings give you a lot of flexibility for developing a connection management policy that meets the needs of your application. As application requirements can vary greatly, we provide recommendations for several common use cases in the subsections below. You should also take the following general guidelines into consideration when developing your policy:

- Connection semantics
  Is your application dependent on a connection remaining open? For example, there may be semantics attached to a connection, as in the case of a session in which a peer's identity, or some allocated resources, are valid only as long as the peer's connection remains active. Glacier2 sessions work this way. In this case you'd want to enable heartbeats and configure the timeout so that it is compatible with the session's expiration timeout.
- Network traversal issues
  Depending on your network topology, security requirements, and other application considerations, a connection may be a valuable resource that is expensive or time-consuming to construct and you may not want Ice to discard it lightly. If sending heartbeats is too costly, you may need to disable automatic closure altogether.
- Aggressiveness
  ACM's combination of heartbeats and connection closure options allows your application to detect when something has gone wrong with a peer. Furthermore, the connection closure options provide varying levels of response, from conservative to aggressive. Your application requirements will dictate the necessary behavior.
- Local networks
  When operating in a purely local network in which there are no firewalls to traverse and none of the other considerations listed above are a concern, you can usually allow Ice to open and close connections transparently. In fact, the default configuration is a reasonable starting point.
- Two sides to every connection
  It's important to consider the requirements of both clients *and* servers when designing your connection management policy. For example, if a client and server are using significantly different settings for their ACM timeouts, it can result in surprising and usually undesirable behavior.

To verify that the policy you've configured is behaving as expected, set `Ice.Trace.Network=2` and monitor your log output for connection-related activity.

Bidirectional Connections

A bidirectional connection allows a server to make callback invocations on a client, offering a simple solution to work around the limitations enforced by network firewalls in which the server would otherwise be prevented from establishing a separate connection back to the client. Given that the client's connection to the server represents the server's only path to that client, this connection must remain open as long as the client needs it.

In versions prior to Ice 3.6, our recommendation was to disable ACM completely in both client and server when using bidirectional connections to prevent unintended closure. As of Ice 3.6, there's no harm in enabling ACM connection closure as long as you also enable client-side or server-side heartbeats to prevent the connection from becoming idle during periods of inactivity. Consider the following settings:

```
# Client
Ice.ACM.Close=0      # CloseOff
Ice.ACM.Heartbeat=3  # HeartbeatAlways
Ice.ACM.Timeout=30

# Server
Ice.ACM.Close=4      # CloseOnIdleForceful
Ice.ACM.Heartbeat=0  # HeartbeatOff
Ice.ACM.Timeout=30
```

Here the client will always send heartbeats at regular intervals to keep connections alive. If a connection does become idle, which most likely

would be due to a serious problem with the client, the server forcefully closes the connection regardless of whether any dispatch or invocations are pending at the time.

> You can also reverse the roles: configure the server to send the heartbeats and the client to close forcefully idle connections.

If your application manually configures bidirectional connections (as opposed to the automatic setup provided by Glacier2 connections, for instance), it's not much extra work to configure the ACM settings individually on the connection object if your requirements for bidirectional connections differ from the communicator-wide settings defined by the ACM properties. As an example, an application may not always need to use a connection for bidirectional purposes. As long as a connection is unidirectional, the application might consider it safe to be closed automatically by ACM. Once it transitions to a bidirectional connection, the program that initiated the connection should modify its ACM configuration similar to the client settings shown above.

Managing Sessions

The notion of a session is a common and very useful solution for managing resources associated with a particular client. Briefly, the idea is that a client creates a session, which usually includes an authentication process, and then allocates some resources. The server associates those resources with the client's session and requires the client to keep the session active, using an expiration timer to reclaim a session and its resources if a client abandons the session without formally terminating it. The strategy represents good defensive programming for a server; without such a solution, ill-behaved clients could continue allocating resources indefinitely.

In versions prior to Ice 3.6, we recommended that the client create a background thread that periodically "pinged" the server using an interval based on the server's session expiration time. The thread is no longer necessary as of Ice 3.6, since the ACM heartbeat functionality serves the same purpose. Note however that there are still a couple of considerations:

- The client needs to determine the server's session expiration time. An application might configure this statically, or the value may only be available at run time by calling an Ice operation. In the latter case, the client would need to transfer this value to a corresponding ACM timeout setting, most likely by calling `setACM` on the connection object.
- Using ACM heartbeats to keep a session alive is convenient for clients but presents a problem for server implementations: How does a server know that the connection is still alive? The solution is for the server to call `setCloseCallback` to get a notification of the connection's closure.

The following configuration settings should get you started:

```
# Client
Ice.ACM.Close=0      # CloseOff
Ice.ACM.Heartbeat=3  # HeartbeatAlways
Ice.ACM.Timeout=30

# Server
Ice.ACM.Close=4      # CloseOnIdleForceful
Ice.ACM.Heartbeat=0  # HeartbeatOff
Ice.ACM.Timeout=30
```

These settings assume that the application configures the ACM timeout statically. (For example, perhaps the ACM timeout also serves directly as the session expiration timeout.) The server's setting for ACM connection closure represents a design decision that assists the implementation: When Ice detects that a connection has become idle, likely due to a serious problem with the client, it forcefully closes the connection. This process involves calling the callback that the server previously set on the connection; presumably the server interprets this notification as an indication that the session can be destroyed. Without the connection callback, the server would have to implement some other strategy for periodically reaping expired sessions.

> In this example, the client has the active role (it sends heartbeats) and the server has the passive role (it expects the client to send heartbeats). It's fine to also reverse these roles. You could for instance configure the client to forcefully close the connection when it becomes idle and configure the server to send heartbeats. The server will no longer receive heartbeats but it will still be reliably notified when the connection is closed if it has registered a close callback.

Detecting Peer Problems

ACM is an effective tool for detecting and recovering from catastrophic problems with a peer. Let's suppose that it's safe to enable ACM connection closure in an application for both the client and the server. Furthermore, in this application, the server can take a significant amount of time to dispatch a client invocation. The client is willing to wait as long as it takes for the invocation to complete, however, as a defensive measure, the client also wants the ability to recover in case the server becomes unresponsive. Consider these settings:

```
# Client
Ice.ACM.Close=3      # CloseOnInvocationAndIdle
Ice.ACM.Heartbeat=0  # HeartbeatOff
Ice.ACM.Timeout=30

# Server
Ice.ACM.Close=1      # CloseOnIdle
Ice.ACM.Heartbeat=1  # HeartbeatOnDispatch
Ice.ACM.Timeout=30
```

The server configuration causes it to close connections when they become idle. Since the client doesn't send heartbeats, this means the client hasn't actively used the connection during the timeout period. The server configuration also sends heartbeats to the client, but only while the server is dispatching invocations. As a result, no matter how long it takes to dispatch the invocations, the Ice run time in the server will continue sending heartbeat messages as an indicator to the client that the server is still "healthy". If the Ice run time in the client determines that a connection has become idle, it either means the connection is no longer being used, or if outgoing invocations are still pending, it means that the server has stopped sending heartbeats for some reason. In the former case, the connection is transparently closed without affecting the client. In the latter case, Ice forcefully closes the connection. Assuming the subsequent automatic retry behavior fails, all pending client invocations on that connection will terminate with an exception.

> Don't use ACM timeouts as a mechanism for aborting long-running invocations. Ice provides invocation timeouts for that purpose.

Disabling ACM

An application may be interested in preventing ACM from closing connections. For example, oneway invocations can be silently discarded when a server closes a connection. One solution is to set the following property in the server:

```
Ice.ACM.Server.Close=0
```

This setting prevents the server from closing an incoming connection regardless of how long the connection has been idle.

Another solution is to enable heartbeats in the client so that the connection remains active and never becomes eligible for closure while the client process is operating normally. If the client should terminate unexpectedly, the server will still be able to close the connection within a reasonable amount of time without waiting for a low-level notification from the network layer.

See Also

- Ice.ACM.*
- Oneway Invocations
- Object Adapters
- Batched Invocations
- Using Connections
- Connection Closure
- Bidirectional Connections
- Glacier2

Using Connections

Applications can gain access to an Ice object representing an established connection.

On this page:

The `Connection` Interface

The Slice definition of the `Connection` interface is shown below:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ConnectionInfo underlying;
        bool incoming;
        string adapterName;
        string connectionId;
    }

 ["delegate"]
    local interface CloseCallback
    {
        void closed(Connection con);
    }

 ["delegate"]
    local interface HeartbeatCallback
    {
        void heartbeat(Connection con);
    }

    local enum ACMClose
    {
        CloseOff,
        CloseOnIdle,
        CloseOnInvocation,
        CloseOnInvocationAndIdle,
        CloseOnIdleForceful
    }
```

```
    local enum ACMHeartbeat
    {
        HeartbeatOff,
        HeartbeatOnInvocation,
        HeartbeatOnIdle,
        HeartbeatAlways
    }

    local struct ACM
    {
        int timeout;
        ACMClose close;
        ACMHeartbeat heartbeat;
    }

    local enum ConnectionClose
    {
        Forcefully,
        Gracefully,
        GracefullyWithWait
    }

    local interface Connection
    {
        void close(ConnectionClose mode);
        Object* createProxy(Identity id);
        void setAdapter(ObjectAdapter adapter);
        ObjectAdapter getAdapter();
        Endpoint getEndpoint();
        void flushBatchRequests();
        void setCloseCallback(CloseCallback callback);
        void setHeartbeatCallback(HeartbeatCallback callback);
        void setACM(optional(1) int timeout, optional(2) ACMClose close,
optional(3) ACMHeartbeat heartbeat);
        ACM getACM();
        string type();
        int timeout();
        string toString();
        ConnectionInfo getInfo();
        void setBufferSize(int rcvSize, int sndSize);
        void throwException();
    }

    local class IPConnectionInfo extends ConnectionInfo
    {
        string localAddress;
        int localPort;
        string remoteAddress;
        int remotePort;
    }
```

```
    local class TCPConnectionInfo extends IPConnectionInfo
    {
        int rcvSize;
        int sndSize;
    }

    local class UDPConnectionInfo extends IPConnectionInfo
    {
        string mcastAddress;
        int mcastPort;
        int rcvSize;
        int sndSize;
    }

    dictionary<string, string> HeaderDict;

    local class WSConnectionInfo extends ConnectionInfo
    {
     HeaderDict headers;
    }
}

module IceSSL
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string cipher;
        Ice::StringSeq certs;
        bool verified;
    }
}

module IceBT
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string localAddress = "";
        int localChannel = -1;
        string remoteAddress = "";
        int remoteChannel = -1;
        string uuid = "";
    }
}

module IceIAP
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string name;
```

```
        string manufacturer;
        string modelNumber;
        string firmwareRevision;
        string hardwareRevision;
```

```
        string protocol;
    }
}
```

As indicated in the Slice definition, a connection is a local interface, similar to a communicator or an object adapter. A connection therefore is only usable within the process and cannot be accessed remotely.

The `Connection` interface supports the following operations:

- `void close(ConnectionClose mode)`
  Explicitly closes the connection using the given closure mode.
- `Object* createProxy(Identity id)`
  Creates a special proxy that only uses this connection. This operation is primarily used for bidirectional connections.
- `void setAdapter(ObjectAdapter adapter)`
  Associates this connection with an object adapter to enable a bidirectional connection.
- `ObjectAdapter getAdapter()`
  Returns the object adapter associated with this connection, or nil if no association has been made.
- `Endpoint getEndpoint()`
  Returns an `Endpoint` object.
- `void flushBatchRequests()`
  Flushes any pending batch requests for this connection.
- `void setCloseCallback(CloseCallback callback)`
  Associates a callback with this connection that is invoked whenever the connection is closed. Passing a nil value clears the current callback.
- `void setHeartbeatCallback(HeartbeatCallback callback)`
  Associates a callback with this connection that is invoked whenever the connection receives a heartbeat message. Passing a nil value clears the current callback.
- `void setACM(optional(1) int timeout, optional(2) ACMClose close, optional(3) ACMHeartbeat heartbeat)`
  Configures Active Connection Management settings for this connection. All arguments are optional, therefore you can change some of the settings while leaving the others unaffected. Refer to your language mapping for more details on optional parameters.
- `ACM getACM()`
  Returns the connection's current settings for Active Connection Management.
- `string type()`
  Returns the connection type as a string, such as `"tcp"`.
- `int timeout()`
  Returns the timeout value used when the connection was established.
- `string toString()`
  Returns a readable description of the connection.
- `ConnectionInfo getInfo()`
  This operation returns a `ConnectionInfo` instance. . Note that the object returned by `getInfo` implements a more derived interface, depending on the connection type. You can down-cast the returned class instance and access the connection-specific information according to the type of the connection.

  The `incoming` member of the `ConnectionInfo` instance is true if the connection is an incoming connection and false, otherwise. If `incoming` is true, `adapterName` provides the name of the object adapter that accepted the connection. The `connectionId` me mber contains the identifier set with the proxy `ice_connectionId` method.

  The `underlying` member contains the underlying transport information if the connection uses a transport that delegates to an underlying transport. For example, the SSL transport delegates to the TCP transport so the `underlying` data member of an SSL connection information is set to a `TCPConnectionInfo` instance. For a WSS connection, `getInfo` returns a `WSConnectionInfo` instance whose `underlying` data member is set to an `IceSSL::ConnectionInfo` whose `underlying` data member is set to a `TCPConnectionInfo`.

- `void setBufferSize(int rcvSize, int sndSize)`
  Sets the connection buffer receive and send sizes.
- `void throwException()`
  Throws an exception indicating the reason for connection closure. For example, `CloseConnectionException` is raised if the connection was closed gracefully by the remote peer, whereas `ConnectionManuallyClosedException` is raised if the connection was manually closed locally by the application. This operation does nothing if the connection is not yet closed.

Flushing Batch Requests for a Connection

The `flushBatchRequests` operation blocks the calling thread until any batch requests that are queued for a connection have been successfully written to the local transport. To avoid the risk of blocking, you can also invoke this operation asynchronously.

Since batch requests are inherently oneway invocations, the async `flushBatchRequests` method does not support a request callback. However, you can use the exception callback to handle any errors that might occur while flushing, and the sent callback to receive notification that the batch request has been flushed successfully.

For example, the code below demonstrates how to flush batch requests asynchronously in C++:

C++11 C++98

```
void flushConnection(Ice::CompressBatch compressBatch,
const std::shared_ptr<Ice::Connection>& conn)
{
    // std::future version also available
    conn->flushBatchRequestsAsync(compressBatch,
                                  [](std::exception_ptr) { cout <<
"Flush failed" << endl; },
                                  [](bool) { cout << "Batch sent" <<
endl; });
}
```

```
class FlushCallback : public IceUtil::Shared
{
public:

    void exception(const Ice::Exception& ex)
    {
        cout << "Flush failed: " << ex << endl;
    }

    void sent(bool sentSynchronously)
    {
        cout << "Batch sent!" << endl;
    }
};
typedef IceUtil::Handle<FlushCallback> FlushCallbackPtr;

void flushConnection(Ice::CompressBatch compressBatch,
const Ice::ConnectionPtr& conn)
{
    FlushCallbackPtr f = new FlushCallback;
    Ice::Callback_Connection_flushBatchRequestsPtr cb =
        Ice::newCallback_Connection_flushBatchRequests(
            f, &FlushCallback::exception, &FlushCallback::sent);
    conn->begin_flushBatchRequests(compressBatch, cb);
}
```

For more information on asynchronous invocations, please see the relevant language mapping chapter.

The `Endpoint` Interface

The `Connection::getEndpoint` operation returns an interface of type `Endpoint`:

Slice

```
module Ice
{
    const short TCPEndpointType = 1;
    const short UDPEndpointType = 3;
    const short WSEndpointType = 4;
    const short WSSEndpointType = 5;
    const short BTEndpointType = 6;
    const short BTSEndpointType = 7;
    const short iAPEndpointType = 8;
    const short iAPSEndpointType = 9;


    local class EndpointInfo
```

```
    {
        EndpointInfo underlying;
        int timeout;
        bool compress;
        short type();
        bool datagram();
        bool secure();
    }

    local interface Endpoint
    {
        EndpointInfo getInfo();
        string toString();
    }

    local class IPEndpointInfo extends EndpointInfo
    {
        string host;
        int port;
        string sourceAddress;
    }

    local class TCPEndpointInfo extends IPEndpointInfo {};

    local class UDPEndpointInfo extends IPEndpointInfo
    {
        byte protocolMajor;
        byte protocolMinor;
        byte encodingMajor;
        byte encodingMinor;
        string mcastInterface;
        int mcastTtl;
    }

    local class WSEndpointInfo extends EndpointInfo
    {
        string resource;
    }

    local class OpaqueEndpointInfo extends EndpointInfo
    {
        Ice::EncodingVersion rawEncoding;
        Ice::ByteSeq rawBytes;
    }
}

module IceSSL
{
    local class EndpointInfo extends Ice::EndpointInfo {};
}
```

```
module IceBT
{
    local class EndpointInfo extends Ice::EndpointInfo
    {
        string addr;
        string uuid;
    }
}

module IceIAP
{
    local class EndpointInfo extends Ice::EndpointInfo
    {
        string manufacturer;
        string modelNumber;
        string name;
```

```
        string protocol;
    }
}
```

The `getInfo` operation returns an `EndpointInfo` instance. Note that the object returned by `getInfo` implements a more derived interface, depending on the endpoint type. You can down-cast the returned class instance and access the endpoint-specific information according to the type of the endpoint, as returned by the `type` operation.

The `timeout` member provides the timeout in milliseconds. The `compress` member is true if the endpoint uses compression (if available). The `datagram` operation returns true if the endpoint is for a datagram transport, and the `secure` operation returns true if the endpoint uses SSL.

The `underlying` member contains the underlying endpoint information if the transport delegates to an underlying transport. For example, the SSL transport uses the TCP transport so the `underlying` data member of an SSL endpoint information is set to a `TCPEndpointInfo` instance. For a WSS endpoint, `getInfo` returns a `WSEndpointInfo` instance whose `underlying` data member is set to an `IceSSL::EndpointInfo` whose `underlying` data member is set to a `TCPEndpointInfo`.

The derived classes provide further detail about the endpoint according to its type.

Opaque Endpoints

An application may receive a proxy that contains an endpoint whose type is unrecognized by the Ice run time. In this situation, Ice preserves the endpoint in its encoded (*opaque*) form so that the proxy remains intact, but Ice ignores the endpoint for all connection-related activities. Preserving the endpoint allows an application to later forward that proxy with all of its original endpoints to a different program that might support the endpoint type in question.

Although a connection will never return an opaque endpoint, it is possible for a program to encounter an opaque endpoint when iterating over the endpoints returned by the proxy method `ice_getEndpoints`.

As a practical example, consider a program for which the IceSSL plug-in is not configured. If this program receives a proxy containing an SSL endpoint, Ice treats it as an opaque endpoint such that calling `getInfo` on the endpoint object returns an instance of `OpaqueEndpointInfo`.

Note that the `type` operation of the `OpaqueEndpointInfo` object returns the *actual* type of the endpoint. For example, the operation returns the value 2 if the object encodes an SSL endpoint. As a result, your program cannot assume that an `EndpointInfo` object whose type is 2 can be safely down-cast to `IceSSL::EndpointInfo`; if the IceSSL plug-in is not configured, such a down-cast will fail because the object is actually an instance of `OpaqueEndpointInfo`.

Client-Side Connection Usage

Clients obtain a connection by using one of the proxy methods `ice_getConnection` or `ice_getCachedConnection`. If the proxy does not yet have a connection, the `ice_getConnection` method immediately attempts to establish one. As a result, the caller must be prepared for this method to block and raise connection failure exceptions. (Use the asynchronous version of this method to avoid blocking.) If the proxy denotes a collocated object and collocation optimization is enabled, calling `ice_getConnection` returns null.

If you wish to obtain the proxy's connection without the potential for triggering connection establishment, call `ice_getCachedConnection`; this method returns null if the proxy is not currently associated with a connection or if connection caching is disabled for the proxy.

As an example, the C++ code below illustrates how to obtain a connection from a proxy and print its type:

C++11 C++98

```
auto proxy = ...
try
{
    auto conn = proxy->ice_getConnection();
    if(conn)
    {
        cout << conn->type() << endl;
    }
    else
    {
        cout << "collocated" << endl;
    }
}
catch(const Ice::LocalException& ex)
{
    cout << ex << endl;
}
```

```
Ice::ObjectPrx proxy = ...
try
{
    Ice::ConnectionPtr conn = proxy->ice_getConnection();
    if(conn)
    {
        cout << conn->type() << endl;
    }
    else
    {
        cout << "collocated" << endl;
    }
}
catch(const Ice::LocalException& ex)
{
    cout << ex << endl;
}
```

Server-Side Connection Usage

Servers can access a connection via the `con` member of the `Ice::Current` parameter passed to every operation. For collocated invocations, `con` has a nil value.

For example, this Java code shows how to invoke `toString` on the connection:

Java

```
public int add(int a, int b, Current current)
{
    if(current.con != null)
    {
        System.out.println("Request received on connection:\n" +
current.con.toString());
    }
    else
    {
        System.out.println("collocated invocation");
    }
    return a + b;
}
```

Although the mapping for the Slice operation `toString` results in a Java method named `_toString`, the Ice run time implements `toString` to return the same value.

Closing a Connection

Applications should rarely need to close a connection manually, but those that do must be aware of its implications. The `ConnectionClose` enumeration defines three closure modes:

- Forcefully
- Gracefully
- Gracefully with wait

Forcefully

A forceful closure causes the peer to receive a `ConnectionLostException`.

A client must use caution when forcefully closing a connection. Any outgoing requests that are pending on the connection when `close` is invoked will fail with a `ConnectionManuallyClosedException`. Furthermore, requests that fail with this exception are not automatically retried.

In a server context, forceful closure can be useful as a defense against hostile clients.

Gracefully

This mode initiates graceful connection closure and causes the local Ice run time to send a `CloseConnection` message to the peer. Any outgoing requests that are pending on the connection when `close` is invoked will fail with a `ConnectionManuallyClosedException`. Furthermore, requests that fail with this exception are not automatically retried.

In a server context, closing a connection gracefully causes Ice to discard any subsequent incoming requests; these requests should eventually be retried automatically when the client receives a `CloseConnection` message. The Ice run time in the server does not send the `CloseConnection` message until all pending dispatched requests have completed.

> After invoking `close(CloseGracefully)`, Ice considers the connection to be in a *closing* state until the remote peer completes its part of the graceful connection closure process. The connection could remain in this state for some time if the peer has no thread pool threads available to process the `CloseConnection` message, and this can prevent operations such as `Communicator::destroy` from completing in a timely manner. Ice uses a timeout to limit the amount of time it waits for a connection to be closed properly. Refer to `Ice.Override.CloseTimeout` for more information.

Gracefully with Wait

In a client context, this mode waits until all pending requests complete before initiating graceful closure. The call to `close` can block indefinitely until the pending requests have completed.

In a server context, closing a connection gracefully causes Ice to discard any subsequent incoming requests; these requests should eventually be retried automatically when the client receives a `CloseConnection` message. The Ice run time in the server does not send the `CloseConnection` message until all pending dispatched requests have completed.

> After invoking `close(CloseGracefullyWithWait)`, Ice considers the connection to be in a *closing* state until the remote peer completes its part of the graceful connection closure process. The connection could remain in this state for some time if the peer has no thread pool threads available to process the `CloseConnection` message, and this can prevent operations such as `Communicator::destroy` from completing in a timely manner. Ice uses a timeout to limit the amount of time it waits for a connection to be closed properly. Refer to `Ice.Override.CloseTimeout` for more information.

See Also

- The Current Object
- Automatic Retries
- Connection Establishment
- Connection Closure
- Bidirectional Connections
- IceSSL
- IceBT
- IceIAP

Connection Closure

The Ice run time may close a connection for many reasons, including the situations listed below:

- When deactivating an object adapter or shutting down a communicator
- As required by active connection management
- When initiated by an application
- After a network operation times out
- In response to an exception, such as a socket failure or protocol error

In most cases, the Ice run time closes a connection gracefully as required by the Ice protocol. The Ice run time only closes a connection forcefully when a timeout or protocol error occurs, or when the application explicitly requests it.

On this page:

Graceful Connection Closure

Gracefully closing a connection occurs in stages:

- In the process that initiates closure, incoming and outgoing requests that are in progress may or may not be allowed to complete, and then a close connection message is sent to the peer. Any incoming requests received after closure is initiated are silently discarded (but may be retried, as discussed in the next bullet). An attempt to make a new outgoing request on the connection results in a `CloseConnectionException` and an automatic retry (if enabled).
- Upon receipt of a close connection message, the Ice run time in the peer closes its end of the connection. Any outgoing requests still pending on that connection fail with a `CloseConnectionException`. This exception indicates to the Ice run time that it is safe to retry those requests without violating at-most-once semantics, assuming automatic retries have not been disabled.
- After detecting that the peer has closed the connection, the initiating Ice run time closes the connection.

Connection Closure and Oneway Invocations

Oneway invocations are generally considered reliable because they are sent over a stream-oriented transport. However, it is quite possible for oneway requests to be silently discarded if a server has initiated graceful connection closure. Whereas graceful closure causes a discarded twoway request to receive a `CloseConnectionException` and eventually be retried, the sender receives no notice about a discarded oneway request.

If an application makes assumptions about the reliability of oneway requests, it may be necessary to control the events surrounding connection closure as much as possible, for example by disabling active connection management and avoiding explicit connection closures.

See Also

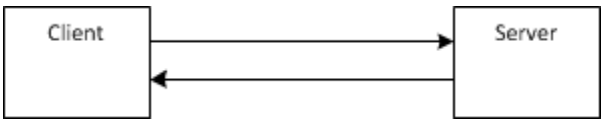
- Active Connection Management
- Using Connections
- Protocol Messages
- Connection Establishment
- Oneway Invocations
- Automatic Retries

Bidirectional Connections

On this page:

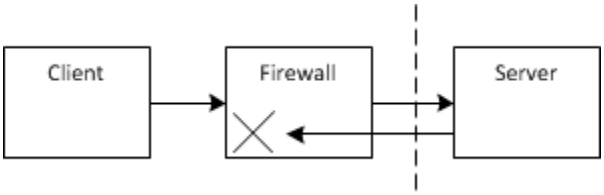
Use Cases for Bidirectional Connections

An Ice connection normally allows requests to flow in only one direction. If an application's design requires the server to make callbacks to a client, the server usually establishes a new connection to that client in order to send callback requests, as shown below:



Callbacks in an open network.

Unfortunately, network restrictions often prevent a server from being able to create a separate connection to the client, such as when the client resides behind a firewall as shown here:



Callbacks with a firewall.

In this scenario, the firewall blocks any attempt to establish a connection directly to the client.

For situations such as these, a bidirectional connection offers a solution. Requests may flow in both directions over a bidirectional connection, enabling a server to send callback requests to a client over the client's existing connection to the server.

There are two ways to make use of a bidirectional connection. First, you can use a Glacier2 router, in which case bidirectional connections are used automatically. If you do not require the functionality offered by Glacier2 or you do not want an intermediary service between clients and servers, you can configure bidirectional connections manually.

The remainder of this section discusses manual configuration of bidirectional connections. An example that demonstrates how to configure a bidirectional connection is provided in the directory `demo/Ice/bidir` of your Ice distribution.

Configuring a Client for Bidirectional Connections

A client needs to perform the following steps in order to configure a bidirectional connection:

1. Create an object adapter to receive callback requests. This adapter does not require a name or endpoints if its only purpose is to receive callbacks over bidirectional connections.
2. Register the callback object with the object adapter, which returns a proxy for this callback object.
3. Obtain a connection object by calling `ice_getConnection` (or `ice_getCachedConnection`) on the proxy, and invoke `setAdapter` on the connection, passing the callback object adapter. This associates the object adapter with the connection and enables callback requests to be dispatched.
4. Pass the proxy to the callback object (obtained in step 2) to the server.

The object adapter remains a regular object adapter, unaware of the connection(s) associated with it through `setAdapter`. These connections have no effect on the endpoints and other properties of proxies created by this object adapter.

However, calling `activate` on this object adapter is optional if it's used only for bidirectional and collocated dispatches.

The code below illustrates these steps:

C++11 C++98 C# Java JavaScript Python

```
auto adapter = communicator->createObjectAdapter("");
auto cbPrx =
Ice::uncheckedCast<CallbackPrx>(adapter->addWithUUID(std::make_shared<C
allbackI>()));
proxy->ice_getCachedConnection()->setAdapter(adapter);
proxy->addClient(cbPrx);
```

```
Ice::ObjectAdapterPtr adapter = communicator->createObjectAdapter("");
CallbackPrx cbPrx = CallbackPrx::uncheckedCast(adapter->addWithUUID(new
CallbackI));
proxy->ice_getCachedConnection()->setAdapter(adapter);
proxy->addClient(cbPrx);
```

```
var adapter = communicator.createObjectAdapter("");
var cbPrx = CallbackPrxHelper.uncheckedCast(adapter.addwithUUID(new
CallbackI()));
proxy.ice_getCachedConnection().setAdapter(adapter);
proxy.addClient(cbPrx);
```

```
com.zeroc.Ice.ObjectAdapter adapter =
communicator.createObjectAdapter("");
CallbackPrx cbPrx = CallbackPrx.uncheckedCast(adapter.addwithUUID(new
CallbackI()));
proxy.ice_getCachedConnection().setAdapter(adapter);
proxy.addClient(cbPrx);
```

```
const adapter = await communicator.createObjectAdapter("");
const cbPrx = CallbackPrx.uncheckedCast(adapter.addwithUUID(new
CallbackI()));
proxy.ice_getCachedConnection().setAdapter(adapter);
await proxy.addClient(cbPrx);
```

```
adapter = communicator.createObjectAdapter("")
cbPrx = CallbackPrx.uncheckedCast(adapter.addwithUUID(CallbackI()))
proxy.ice_getCachedConnection().setAdapter(adapter)
proxy.addClient(cbPrx)
```

The proxy to the callback object (`cbPrx` in the code above) is a regular proxy, with no endpoints. Even if it had endpoints, we would not want the server to use these endpoints–we want the server to reuse the already established connection. As we will see below, the server will convert this callback proxy into a new fixed proxy bound to the connection.

Active Connection Management Considerations

Active Connection Management (ACM) automatically and transparently closes idle connections. As far as the Ice run time is concerned, an outgoing connection is a client connection regardless of whether it's also used as a bidirectional connection, therefore the client-side ACM properties govern the ACM behavior for bidirectional connections in a client. Also note that the client's outgoing connection is the only channel on which the server can send callback invocations to the client, therefore prematurely closing the connection (such as allowing the ACM facility to close it automatically) introduces the risk that the client might unknowingly fail to receive callbacks.

It is not necessary to disable client-side ACM when using bidirectional connections. If you leave ACM enabled, you simply need to ensure that the connection remains active to prevent the ACM facility from closing it. The easiest way to accomplish this is to enable connection heartbeats:

```
Ice.ACM.Client.Heartbeat=Always
```

Alternatively, you can enable heartbeats on a bidirectional connection by calling `setACM`:

C++11 C++98 C# Java JavaScript Python

```
proxy->ice_getCachedConnection()->setACM(Ice::nullopt, Ice::nullopt,
Ice::ACMHeartbeat::HeartbeatAlways);
```

```
proxy->ice_getCachedConnection()->setACM(IceUtil::None, IceUtil::None,
Ice::HeartbeatAlways);
```

```
proxy.ice_getCachedConnection().setACM(Ice.Util.None, Ice.Util.None,
Ice.ACMHeartBeat.HeartbeatAlways);
```

```
proxy.ice_getCachedConnection().setACM(null, null,
Optional.of(com.zeroc.Ice.ACMHeartbeat.HeartbeatAlways));
```

```
proxy.ice_getCachedConnection().setACM(undefined, undefined,
Ice.ACMHeartbeat.HeartbeatAlways);
```

```
proxy.ice_getCachedConnection().setACM(Ice.Unset, Ice.Unset,
Ice.ACMHeartbeat.HeartbeatAlways)
```

The server's ACM configuration also plays an important role here. For more information, refer to our discussion of ACM configurations for bidirectional connections.

Configuring a Server for Bidirectional Connections

A server needs to take the following steps in order to make callbacks over a bidirectional connection:

1. Obtain a proxy to the callback object; this proxy is typically supplied by the client.
2. Convert this proxy into a fixed proxy bound to the connection, by calling the `ice_fixed` proxy method with the connection object. The connection object is accessible as a member of the `Ice::Current` parameter supplied to an operation implementation.

These steps are illustrated in the C++ code below:

C++11 C++98 C# Java Python

```
void addClient(shared_ptr<ClientPrx> client,
const Ice::Current& current)
{
    client->ice_fixed(current.con)->notify();
}
```

```
void addClient(const ClientPrx& client, const Ice::Current& current)
{
    client->ice_fixed(current.con)->notify();
}
```

```
public override void addClient(ClientPrx client, Ice.Current current)
{
    ((ClientPrx)client.ice_fixed(current.con)).notify();
}
```

```
public void addClient(ClientPrx client, com.zeroc.Ice.Current current)
{
    client.ice_fixed(current.con).notify();
}
```

```
def addClient(self, client, current):
    client.ice_fixed(current.con).notify()
```

If the client forgot to call `setAdapter` on the connection, the `notify` call on the fixed proxy fails with an `ObjectNotExistException` raised by the client and propagated to the server.

Nested call
With this example, the client thread pool of the client can have only one thread unless the implementation of `notify` makes a remote call and waits for its result.

The server's server thread pool must have at least two threads if `notify` is a two-way call: one thread that dispatches `addClient`, and another thread that receives the (void) result of the `notify` call. A single thread in the server thread pool would result in a thread-starvation deadlock.

Fixed Proxies

The proxy returned by the proxy method `ice_fixed` is called a *fixed proxy*. It cannot be marshaled; attempts to do so raise `FixedProxyException`. The connection's `createProxy` operation also returns a fixed proxy.

A fixed proxy is bound to the connection that created it, and ceases to work once that connection is closed. If the connection is closed prematurely, either by active connection management (ACM) or by explicit action on the part of the application, the server can no longer make callback requests using that proxy. Any attempt to use the proxy again usually results in a `CloseConnectionException`.

Many aspects of a fixed proxy cannot be changed. For example, it is not possible to change the proxy's endpoints or timeout. Attempting to invoke a method such as `ice_timeout` on a fixed proxy raises `FixedProxyException`.

A fixed proxy created by `ice_fixed` inherits all relevant aspects of the parent proxy, such as its context and its compression and secure settings. A fixed proxy created by `createProxy` on a connection won't use compression unless `Ice.Override.Compress=1` is set. To enable compression without `Ice.Override.Compress=1`, you can call `ice_compress(true)` on the fixed proxy to create a new fixed proxy with compression enabled.

Limitations of Bidirectional Connections

Bidirectional connections have certain limitations:

- They can only be configured for connection-oriented transports such as TCP and SSL.
- Most proxy factory methods are not relevant for a fixed proxy. The proxy is bound to an existing connection, therefore the proxy reflects the connection's configuration. Attempting to change settings such as the proxy's timeout value causes the Ice run time to raise `FixedProxyException`. Note however that it is legal to configure a fixed proxy for using oneway or twoway invocations. You may also invoke `ice_secure` on a fixed proxy if its security configuration is important; a fixed proxy configured for secure communication raises `NoEndpointException` on the first invocation if the connection is not secure.
- A connection established from a Glacier2 router to a server is not configured for bidirectional use. Only the connection from a client to the router is bidirectional. However, the client must not attempt to manually configure a bidirectional connection to a router, as this is handled internally by the Ice run time.

Threading Considerations for Bidirectional Connections

The Ice run time normally creates two thread pools for processing network traffic on connections: the client thread pool manages outgoing connections and the server thread pool manages incoming connections. All of the object adapters in a server share the same thread pool by default, but an object adapter can also be configured to have its own thread pool. The default size of the client and server thread pools is one.

The client thread pool processes replies to pending requests. When a client configures an outgoing connection for bidirectional requests, the client thread pool also becomes responsible for dispatching callback requests received over that connection. Similarly, the server thread pool normally dispatches requests from clients. If a server uses a bidirectional connection to send callback requests, then the server thread pool must also process the replies to those requests.

You must increase the size of the appropriate thread pool if you need the ability to dispatch multiple requests in parallel, or if you need to make nested twoway invocations. For example, a client that receives a callback request over a bidirectional connection and makes nested invocations must increase the size of the *client* thread pool.

See Also

- Glacier2
- Creating an Object Adapter
- Servant Activation and Deactivation

Connection Timeouts

On this page:

Overview of Connection Timeouts

> Connection timeouts only affect network operations. Use invocation timeouts to limit the amount of time a client waits for an operation to complete.

Connection timeouts allow applications to detect low-level network problems in a reasonable period of time. Ice enforces connection timeouts when performing network operations such as establishing a connection, reading and writing to a connection, and closing a connection. Disabling connection timeouts means an application may not discover a network issue until much later (if at all) when low-level network protocols finally detect and report the problem, therefore Ice enables connection timeouts by default and we strongly encourage you to use them.

You should normally choose your connection timeouts based on the speed of the network on which the connections take place. For example, the connection timeout for a local gigabit network will usually be much smaller than the timeout for a 56kbps connection.

Finally, note that timeouts in Ice are "soft" timeouts, in the sense that they are not precise, real-time timeouts. (The precision is limited by the capabilities of the underlying operating system.)

Configuring Connection Timeouts

Ice supports several configuration properties that you can use to control connection timeouts:

- `Ice.Default.Timeout`
  This property specifies the default timeout in milliseconds for all connections. This property's default value of `60000` means connection timeouts are enabled by default with a timeout of 60 seconds.
- `Ice.Override.ConnectTimeout`
  This setting overrides any existing connection timeout, but only applies while Ice establishes a connection.
- `Ice.Override.CloseTimeout`
  This setting overrides any existing connection timeout, but only applies while Ice closes a connection.
- `Ice.Override.Timeout`
  This setting overrides any existing connection timeout. It applies to all network operations unless superseded by `Ice.Override.ConnectTimeout` or `Ice.Override.CloseTimeout`.

You can also configure connection timeouts individually for the endpoints of a proxy. Consider this example:

```
MyProxy=hello:tcp -h 10.0.0.1 -t 1000:tcp -h 205.125.53.4 -t 5000
```

This stringified proxy contains two TCP endpoints. The first endpoint refers to a host in a local network and its `-t` option specifies a connection timeout of one second. The second endpoint refers to a host on the internet and uses a connection timeout of five seconds. If a proxy's endpoints don't include timeouts, the endpoints inherit the communicator's default timeout set by `Ice.Default.Timeout`. The various `Ice.Override` properties take precedence over any endpoint timeouts.

Finally, the proxy method `ice_timeout` returns a new proxy in which all of its endpoints have the specified timeout:

C++11 C++98

```
auto p = communicator->stringToProxy("hello:tcp -h 10.0.0.1 -t 1000:tcp
-h 205.125.53.4 -t 5000");

cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t
1000:tcp -h 205.125.53.4 -t 5000
p = p->ice_timeout(1500);
cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t
1500:tcp -h 205.125.53.4 -t 1500
```

```
Ice::ObjectPrx p = communicator->stringToProxy("hello:tcp -h 10.0.0.1 -t
1000:tcp -h 205.125.53.4 -t 5000");

cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t
1000:tcp -h 205.125.53.4 -t 5000
p = p->ice_timeout(1500);
cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t
1500:tcp -h 205.125.53.4 -t 1500
```

The `Ice.Override` properties also take precedence over any timeouts configured via `ice_timeout`.

Connection Timeout Failures

Ice considers a connection timeout to indicate a serious error with the underlying network connection, causing Ice to immediately close the connection and report a `TimeoutException`, or one of its subclasses `ConnectTimeoutException` or `CloseTimeoutException` if appropriate, for all of the invocations pending on that connection.

> The application may not receive an exception immediately after a connection timeout occurs, as Ice may attempt the invocation again if automatic retries are enabled and the invocation qualifies for retry. Since its default configuration allows Ice to perform one retry without a delay, it will appear to the application as if the timeout takes twice as long as expected to be reported.

In most cases you won't need to handle a timeout as a special case and can use the typical exception clauses:

C++11 C++98

```
std::shared_ptr<Filesystem::FilePrx> myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

```
Filesystem::FilePrx myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

If your application needs to treat timeouts differently, you can catch them as follows:

C++11 C++98

```
std::shared_ptr<Filesystem::FilePrx> myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::ConnectTimeoutException&)
{
    cerr << "connect timed out!" << endl;
}
catch(const Ice::CloseTimeoutException&)
{
    cerr << "close timed out!" << endl;
}
catch(const Ice::TimeoutException&)
{
    cerr << "timed out!" << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

```
Filesystem::FilePrx myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::ConnectTimeoutException&)
{
    cerr << "connect timed out!" << endl;
}
catch(const Ice::CloseTimeoutException&)
{
    cerr << "close timed out!" << endl;
}
catch(const Ice::TimeoutException&)
{
    cerr << "timed out!" << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

ACM and Timeouts

The Active Connection Management features can also help your application in detecting and dealing with connectivity issues and complement the connection timeout behavior. For example, connection timeouts only have an effect when the Ice run time is actively performing network operations, but it's also possible for problems to arise during periods of inactivity. These problems may not be noticed for some time.

The ACM heartbeat facility can optionally be configured to send heartbeat messages at regular intervals when a connection would otherwise be idle. Doing so makes it more likely that the Ice run time will detect a connectivity issue in a timely manner.

Connection Reuse and Timeouts

Ice tries to reuse existing connections as much as possible unless the application indicates otherwise. When an application makes its first invocation on a proxy, Ice searches its list of open connections to see if any of them match any of the proxy's endpoints. Determining whether an existing connection is "compatible" with a proxy endpoint, and therefore suitable for reuse, involves several criteria. One attribute that Ice considers is the timeout: the endpoint's timeout and the connection's timeout must match for the connection to be considered eligible for reuse.

If you encounter a situation where Ice is opening a new connection when you expect it to use an existing connection, your timeout configuration may be the reason.

See Connection Establishment for more information on how Ice decides whether to open a new connection or reuse an existing one.

See Also

- Proxy Methods
- Invocation Timeouts
- Automatic Retries
- Connection Establishment

Collocated Invocation and Dispatch

On this page:

Overview

One of the useful features of the Ice run time is that it is *location transparent*: the client does not need to know where the implementation of an Ice object resides; an invocation on an object automatically is directed to the correct target, whether the object is implemented in the local address space, in another address space on the same machine, or in another address space on a remote machine. Location transparency is important because it allows us to change the location of an object implementation without breaking client programs and, by using a location service such as IceDiscovery or IceGrid, addressing information such as host names and port numbers can be externalized so they do not appear in stringified proxies.

Collocated Invocations

For invocations that cross address space boundaries (or more accurately, cross communicator boundaries), the Ice run time dispatches requests via the appropriate transport. However, for a proxy invocation in which the proxy and the servant that processes the invocation share the same communicator (so-called *collocated* invocations), the Ice run time, by default, does not send the invocation via the transport specified in the proxy. Instead, collocated invocations take a short-cut inside the Ice run time and are dispatched more efficiently.

> Note that if the proxy and the servant do not use the same communicator, the invocation is *not* collocated, even though caller and callee are in the same address space.

The reason for this is if collocated invocations were sent via TCP/IP, for example, invocations would still be sent via the operating system kernel (using the back plane instead of a network) and would incur the full cost of creating TCP/IP connections, trapping in and out of the kernel, and so on. By optimizing collocated requests, much of this overhead can be avoided.

For efficiency reasons, collocated invocations are not completely location transparent, that is, a collocated call has semantics that differ in some ways from calls that cross address-space boundaries. Specifically, collocated invocations differ from ordinary invocations in the following respects:

- Most collocated invocations are dispatched in the server-side thread pool just like regular invocations; the only exceptions are synchronous twoway collocated invocations with no invocation timeout, which are dispatched in the calling thread.
- The object adapter holding state is ignored: collocated invocations proceed normally even if the target object's adapter is in the holding state.
- AMI callbacks for asynchronous collocated invocations are dispatched from the servant's calling thread and not from the client-side thread pool unless AMD is used for the servant dispatch. In this case, the AMI callback is called from the client-side thread pool.
- Invocation timeouts work as usual, but connection timeouts are ignored.

In practice, these differences rarely matter. The most likely cause of surprises with collocated invocations is dispatch in the calling thread, that is, a collocated invocation behaves like a local, synchronous procedure call. This can cause problems if, for example, the calling thread acquires a lock that an operation implementation tries to acquire as well: unless you use recursive mutexes, this will cause deadlock.

The Ice run time uses the following semantics to determine whether a proxy is eligible for the collocated optimization:

- For an indirect proxy, collocation optimization is used if the proxy's adapter ID matches the adapter ID or replica group ID of an object adapter in the same communicator.
- For a well-known proxy, the Ice run time queries each object adapter Active Servant Map to determine if the servant is local.
- For a direct proxy, the Ice run time performs an endpoint search using the proxy's endpoints.

When an endpoint search is required, the Ice run time compares each of the proxy's endpoints against the endpoints of the communicator's object adapters. Only the transport, address and port are considered; other attributes of an endpoint, such as timeout settings, are not considered during this search. If a match is found, the invocation is dispatched using collocation optimization. Normally this search is executed only once, during the proxy's first invocation, although the proxy's connection caching setting influences this behavior.

Collocation optimization is enabled by default, but you can disable it for all proxies by setting the property `Ice.Default.CollocationOptimized=0`. You can also disable the optimization for an individual proxy using the factory method `ice_collocationOptimized(false)`. Finally, for proxies created from a property using `propertyToProxy`, the property *`name`*`.CollocationOptimized` configures the default setting for the proxy.

Creating an Object Adapter for Collocated Invocations

An object adapter requires no endpoints if its only purpose is to dispatch collocated requests. You can create a "collocated-only" object adapter by calling `createObjectAdapter` with an empty name:

C++11 C++98

```
auto collocAdapter = communicator->createObjectAdapter(""); //
collocAdapter is std::shared_ptr<Ice::ObjectAdapter>
```

```
Ice::ObjectAdapterPtr collocAdapter =
communicator->createObjectAdapter("");
```

Normally Ice requires an object adapter to either have endpoints or be configured with a router, but Ice relaxes this restriction when the name is empty. Note however that this means collocated-only object adapters cannot be configured using properties.

Proxies created by a collocated-only object adapter contain no endpoints, which essentially makes them well-known proxies. When the program makes its initial invocation on such a proxy, Ice searches every local object adapter for a servant with an identity matching that of the proxy. For Ice to successfully find the collocated servant, it must be present in the object adapter's active servant map. Servant locators and default servants are not queried in this situation.

See Also

- IceDiscovery
- IceGrid
- Connection Establishment
- Proxy Methods
- Obtaining Proxies

Locators

In Terminology, we described briefly how the Ice run time uses an intermediary, known as a *location service*, to convert the symbolic information in an indirect proxy into an endpoint that it can use to communicate with a server. This section expands on that introduction to explain in more detail how the Ice run time interacts with a location service. You can create your own location service or you can use IceDiscovery or IceGrid, which are both implementations of a location service. Describing how to implement a location service is outside the scope of this manual.

A *locator* is an Ice object that is implemented by a location service. A locator object must support the Slice interface `Ice::Locator`, which defines operations that satisfy the location requirements of the Ice run time. Applications do not normally use these operations directly, but the locator object may support an implementation-specific interface derived from `Ice::Locator` that provides additional functionality. For example, IceGrid's locator object provides access to an `IceGrid::Query` object so that applications can perform more sophisticated queries.

Topics

See Also

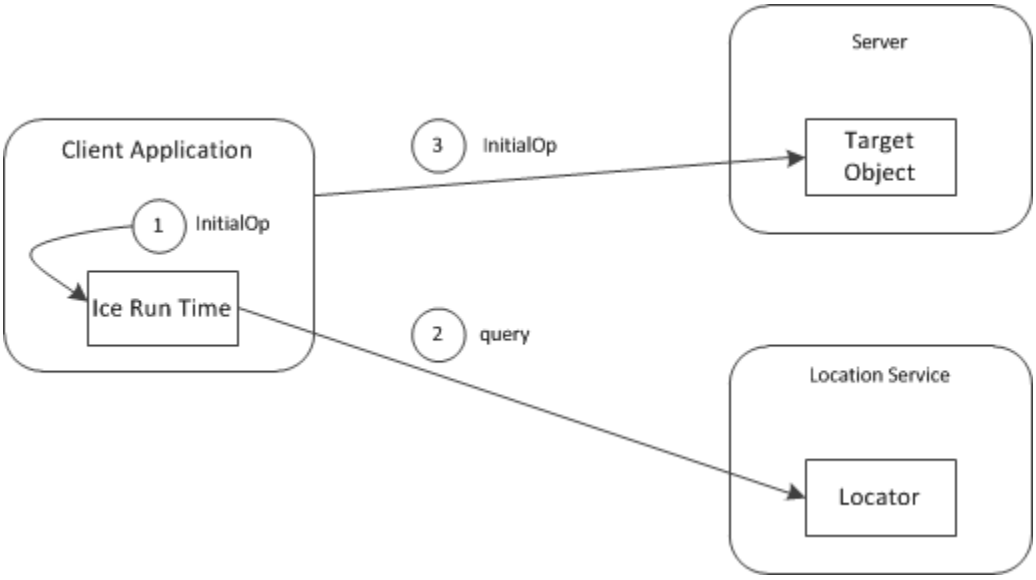
- Terminology
- IceDiscovery
- IceGrid
- Querying Well-Known Objects

Locator Semantics for Clients

On this page:

Invocations with an Indirect Proxy

On the first use of an indirect proxy in an application, the Ice run time may issue a remote invocation on the locator object. This activity is transparent to the application, as shown below:



Locating an object.

1. The client invokes the operation `initialOp` on an indirect proxy.
2. The Ice run time checks an internal cache (called the *locator cache*) to determine whether a query has already been issued for the symbolic information in the proxy. If so, the cached endpoint is used and an invocation on the locator object is avoided. Otherwise, the Ice run time issues a locate request to the locator.
3. If the object is successfully located, the locator returns its current endpoints. The Ice run time in the client caches this information, es tablishes a connection to one of the endpoints, and proceeds to send the invocation as usual.
4. If the object's endpoints cannot be determined, the client receives an exception. `NotRegisteredException` is raised when an identity, object adapter identifier or replica group identifier is not known. A client may also receive `NoEndpointException` if the location service failed to determine the current endpoints.

As far as the Ice run time is concerned, the locator simply converts the information in an indirect proxy into usable endpoints. Whether the locator's implementation is more sophisticated than a simple lookup table is irrelevant to the Ice run time. However, the act of performing this conversion may have additional semantics that the application must be prepared to accept.

For example, when using IceGrid as your location service, the target server may be launched automatically if it is not currently running, and the locate request does not complete until that server is started and ready to receive requests. As a result, the initial request on an indirect proxy may incur additional overhead as all of this activity occurs.

Replication with a Locator

An indirect proxy may substitute a replica group identifier in place of the object adapter identifier. In fact, the Ice run time does not distinguish between these two cases and considers a replica group identifier as equivalent to an object adapter identifier for the purposes of resolving the proxy. The location service implementation must be able to distinguish between replica groups and object adapters using only this identifier.

The location service may return multiple endpoints in response to a locate request for an adapter or replica group identifier. These endpoints might all correspond to a single object adapter that is available at several addresses, or to multiple object adapters each listening at a single address, or some combination thereof. The Ice run time attaches no semantics to the collection of endpoints, but the application can make assumptions based on its knowledge of the location service's behavior.

When a location service returns more than one endpoint, the Ice run time behaves exactly as if the proxy had contained several endpoints. As always, the goal of the Ice run time is to establish a connection to one of the endpoints and deliver the client's request. By default, all requests made via the proxy that initiated the connection are sent to the same server until that connection is closed.

After the connection is closed, such as by Active Connection Management (ACM), subsequent use of the proxy causes the Ice run time to obtain another connection. Whether that connection uses a different endpoint than previous connections depends on a number of factors, but it is possible for the client to connect to a different server than for previous requests.

Locator Cache

After successfully resolving an indirect proxy, the location service must return at least one endpoint. How the service derives the list of endpoints that corresponds to the proxy is entirely implementation dependent. For example, IceGrid's location service can be configured to respond in a variety of ways; one possibility uses a simple round-robin scheme, while another selects endpoints based on the system load of the target hosts.

A locate request has the potential to significantly increase the latency of the application's invocation with a proxy, and this is especially true if the locate request triggers additional implicit actions such as starting a new server process. Fortunately, this overhead is normally incurred only during the application's initial invocation on the proxy, but this impact is influenced by the Ice run time's caching behavior.

To minimize the number of locate requests, the Ice run time caches the results of previous requests. By default, the results are cached indefinitely, so that once the Ice run time has obtained the endpoints associated with an indirect proxy, it never issues another locate request for that proxy. Furthermore, the default behavior of a proxy is to cache its connection, that is, once a proxy has obtained a connection, it continues to use that connection indefinitely.

Taken together, these two caching characteristics represent the Ice run time's best efforts to optimize an application's use of a location service: after a proxy is associated with a connection, all future invocations on that proxy are sent on the same connection without any need for cache lookups, locate requests, or new connections.

If a proxy's connection is closed, the next invocation on the proxy prompts the Ice run time to consult its locator cache to obtain the endpoints from the prior locate request. Next, the Ice run time searches for an existing connection to any of those endpoints and uses that if possible (assuming the proxy has connection caching enabled), otherwise it attempts to establish a new connection to each of the endpoints until one succeeds. Only if that process fails does the Ice run time clear the entry from its cache and issue a new locate request with the expectation that a usable endpoint is returned.

The Ice run time's default behavior is optimized for applications that require minimal interaction with the location service, but some applications can benefit from more frequent locate requests. Normally this is desirable when implementing a load-balancing strategy, as we discuss in more detail below. In order to increase the frequency of locate requests, an application must configure a timeout for the locator cache and manipulate the connections of its proxies.

Locator Cache Timeout

An application can define a timeout to control the lifetime of entries in the locator cache. This timeout can be specified globally using the `Ice.Default.LocatorCacheTimeout` property and for individual proxies using the proxy method `ice_locatorCacheTimeout`. The Ice run time's default behavior is equivalent to a timeout value of `-1`, meaning the cache entries never expire. Using a timeout value greater than zero causes the cache entries to expire after the specified number of seconds. Finally, a timeout value of zero disables the locator cache altogether.

The previous section explained the circumstances in which the Ice run time consults the locator cache. Briefly, this occurs only when the application has invoked an operation on a proxy and the proxy is not currently associated with a connection. If the timeout is set to zero, the Ice run time issues a new locate request immediately. Otherwise, for a non-zero timeout, the Ice run time examines the locator cache to determine whether the endpoints from the previous locate request have expired. If so, the Ice run time discards them and issues a new locate request.

Given this behavior, if your goal is to force a proxy invocation to issue locate requests more frequently, you can do so only when the proxy is not associated with a connection. You can accomplish that in several ways:

- create a new proxy, which is inherently not connected by default
- explicitly close the proxy's existing connection
- disable the proxy's connection caching behavior

Of these choices, the last one is the most common.

Load Balancing with a Locator

Ice supports proxy-based load balancing whose behavior is driven solely by a proxy's configuration settings. A disadvantage of relying solely on this form of load balancing is that the client cannot make any intelligent decisions based on the status of the servers. If you want to distribute your requests in a more sophisticated way, you must either modify your clients to query the servers directly, or use a location service that can transparently direct a client to an appropriate server. For example, the IceGrid location service can monitor the system load on each server host and use that information when responding to locate requests.

The location service may return only one endpoint, which presumably represents the best server (at that moment) for the client to use. With only one endpoint available, changing the proxy's endpoint selection type makes no difference. However, by disabling connection caching and modifying the locator cache timeout, the application can force the Ice run time to periodically retrieve an updated endpoint from the location service. For example, an application can set a locator cache timeout of thirty seconds and communicate with the selected server for that period. After the timeout has expired, the next invocation prompts the Ice run time to issue a new locate request, at which point the client might be directed to a different server.

If the location service returns multiple endpoints, the application must be designed with knowledge of how to interpret them. For instance, the location service may attach semantics to the order of the endpoints (such as least-loaded to most-loaded) and intend that the application use the endpoints in the order provided. Alternatively, the client may be free to select any of the endpoints. As a result, the application and the location service must cooperate to achieve the desired results.

You can combine the simple form of load balancing described in the previous section with an intelligent location service to gain even more flexibility. For example, suppose an application expects to receive multiple endpoints from the location service and has configured its proxy to disable connection caching and set a locator cache timeout. For each invocation, Ice run time selects one of the endpoints provided by the location service. When the timeout expires, the Ice run time issues a new locate request and obtains a fresh set of endpoints from which to choose.

See Also

- Terminology
- Proxy Methods
- Connection Establishment
- Active Connection Management
- Ice.Default.*
- Proxy-Based Load Balancing

Locator Configuration for a Client

An Ice client application must supply a proxy for the locator object, which it can do in several ways:

- by explicitly configuring an indirect proxy using the `ice_locator` proxy method
- by calling `setDefaultLocator` on a communicator, after which all new proxies use the given locator by default
- by defining the `Ice.Default.Locator` configuration property, which causes all proxies to use the given locator by default

The Ice run time's efforts to resolve an indirect proxy can be traced by setting the following configuration properties:

```
Ice.Trace.Network=2
Ice.Trace.Protocol=1
Ice.Trace.Locator=2
```

See Ice.Trace.* for more information on these properties.

See Also

- Locator Semantics for Clients
- Proxy Methods
- Ice.Default.*
- Ice.Trace.*

Locator Semantics for Servers

A location service must know the endpoints of any object adapter whose identifier can be used in an indirect proxy. For example, suppose a client uses the following proxy:

```
Object1@PublicAdapter
```

The Ice run time in the client includes the identifier `PublicAdapter` in its locate request and expects to receive the associated endpoints. The only way the location service can know these endpoints is if it is given them. When you consider that an object adapter's endpoints may not specify fixed ports, and therefore the endpoint addresses may change each time the object adapter is activated, it is clear that the best source of endpoint information is the object adapter itself. As a result, an object adapter that is properly configured contacts the locator during activation to supply its identifier and current endpoints. More specifically, the object adapter registers itself with an object implementing the `Ice::LocatorRegistry` interface, whose proxy the object adapter obtains from the locator.

A location service may require that all object adapters be pre-registered via some implementation-specific mechanism. (IceGrid behaves this way by default.) This implies that activation can fail if the object adapter supplies an identifier that is unknown to the location service. In such a situation, the object adapter's `activate` operation raises `NotRegisteredException`.

In a similar manner, an object adapter that participates in a replica group includes the group's identifier in the locator request that is sent during activation. If the location service requires replica group members to be configured in advance, `activate` raises `NotRegisteredException` if the object adapter's identifier is not one of the group's registered participants.

See Also

- Object Adapters
- Locator Semantics for Clients
- Locator Configuration for a Server
- IceGrid
- Terminology
- Object Adapter Replication

Locator Configuration for a Server

On this page:

Configuring an Object Adapter with a Locator

An object adapter must be able to obtain a locator proxy in order to register itself with a location service. Each object adapter can be configured with its own locator proxy by defining its `Locator` property, as shown in the example below for the object adapter named `SampleAdapter`:

```
SampleAdapter.Locator=IceGrid/Locator:tcp -h locatorhost -p 10000
```

Alternatively, a server may call `setLocator` on the object adapter prior to activation. If the object adapter is not explicitly configured with a locator proxy, it uses the default locator as provided by its communicator.

Two other configuration properties influence an object adapter's interactions with a location service during activation:

- `AdapterId`
  Configuring a non-empty identifier for the `AdapterId` property causes the object adapter to register itself with the location service. A locator proxy must also be configured.
- `ReplicaGroupId`
  Configuring a non-empty identifier for the `ReplicaGroupId` property indicates that the object adapter is a member of a replica group. For this property to have an effect, `AdapterId` must also be configured with a non-empty value.

We can use these properties as shown below:

```
SampleAdapter.AdapterId=SampleAdapterId
SampleAdapter.ReplicaGroupId=SampleGroupId
SampleAdapter.Locator=IceGrid/Locator:tcp -h locatorhost -p 10000
```

Note that a location service may enforce pre-registration requirements.

Registering a Process with a Locator

An activation service, such as an IceGrid node, needs a reliable way to gracefully deactivate a server. One approach is to use a platform-specific mechanism, such as POSIX signals. This works well on POSIX platforms when the server is prepared to intercept signals and react appropriately. On Windows platforms, it works less reliably for C++ servers, and not at all for Java servers. For these reasons, Ice provides an alternative that is both portable and reliable:

Slice

```
module Ice
{
    interface Process
    {
        void shutdown();
        void writeMessage(string message, int fd);
    }
};
```

The Slice interface `Process` allows an activation service to request a graceful shutdown of the server. When `shutdown` is invoked, the object implementing this interface is expected to initiate the termination of its server process. The activation service may expect the server to terminate within a certain period of time, after which it may terminate the server abruptly.

One of the benefits of the Ice administrative facility is that it creates an implementation of `Process` and makes it available via an administrative object adapter, or your own object adapter. Furthermore, IceGrid automatically enables this facility on the servers that it activates.

See Also

- Object Adapters
- Locators
- Locator Configuration for a Client
- Locator Semantics for Servers
- The Process Facet
- Administrative Facility
- IceGrid

Routers

This page describes the Ice router facility.

On this page:

Router Overview

A *router* is an Ice object that provides information to the Ice run time to allow routing messages between clients and servers. In a client, configuring a proxy to use a router produces a *routed proxy* on which all invocations are sent to the router for forwarding to the target server. The endpoints in a routed proxy are ignored by the Ice run time, at least for the purposes of connection establishment. Instead, the Ice run time in the client establishes a connection to the router and may pass along the proxy's endpoints so that the router can establish its own connection to the target server. Ice typically opens only one connection to a router and reuses that connection for invocations on all of the routed proxies configured to use the same router.

The Ice distribution includes two router implementations that serve different purposes:

- Glacier2 is a sophisticated routing service that is commonly used as a single point of entry through which clients on public networks access back-end services on an internal network, with support for application-specific session management and access control.
- IceBridge is a simpler service that facilitates bridging connections over different transports.

Both services implement the `Ice::Router` interface, which the Ice run time requires of any router implementation.

Slice

```
module Ice
{
    interface Router
    {
        ["nonmutating", "cpp:const"] idempotent Object*
getClientProxy(out optional(1) bool hasRoutingTable);

        ["nonmutating", "cpp:const"] idempotent Object*
getServerProxy();

        idempotent ObjectProxySeq addProxies(ObjectProxySeq proxies);
    }

    interface RouterFinder
    {
        Router* getRouter();
    }
}
```

Default Router

A communicator can be configured with a default router. The most common way to configure the default router is to set the property `Ice.Default.Router`. The value of this property is a proxy for the router's primary Ice object, as shown in the example below for Glacier2:

```
Ice.Default.Router=Glacier2/router:tcp -h routerhost -p 4063
```

You can also specify a default router by calling `setDefaultRouter` on the communicator, and obtain the current setting using `getDefaultRouter`.

> Proxies created with this communicator are configured with this router by default (see below). Object adapters created with this communicator are not affected by the default router.

Configuring a Router for Client Invocations

Setting a default router as described above means every proxy created by the communicator will be configured to use the router by default. If your client needs to use a router more selectively, you can use the `ice_router` proxy method to obtain a routed proxy:

C++

```
shared_ptr<Ice::RouterPrx> router = ...;
auto target = communicator->stringToProxy("...");
target = target->ice_router(router);
```

As with all proxy factory methods, `ice_router` returns a new proxy with the requested configuration.

Another way to configure a router is with a proxy property:

```
MyProxy.Router=Glacier2/router:tcp -h routerhost -p 4063
```

In this example, calling `propertyToProxy("MyProxy")` on a communicator returns a proxy that is already configured to use Glacier2.

Configuring a Client for Callbacks

A client that needs to receive callbacks from the server through the router creates an object adapter that hosts callback objects and associates the router with this object adapter. You create this object adapter-router association using the property *`adapter`*`.Router` (in the object adapter's configuration), or by creating the object adapter with `createObjectAdapterWithRouter`.

This is a one-to-one association: an object adapter can be associated with a single router and likewise a router can be associated with only one object adapter.

When the object adapter is created, it calls `setAdapter` on the connection between the client and the router. This connection is either reused (if a connection to the router already existed) or established during the creation of the object adapter. Later, when the server calls on a proxy to the callback object, the router forwards the request to the object adapter over this bidirectional connection between the client and the router.

There are two differences between an object adapter that you configure for bidirectional dispatch (with `setAdapter` on a connection) and an object adapter configured with a router:

- You can call `setAdapter` with the same object adapter on multiple connections, whereas you can have only one router associated with a given object adapter.
- The endpoints of the proxies created by an object adapter with a configured router are the endpoints of the proxy returned by `getServerProxy`, which typically point to the "server-side" of the router. Endpoints and published endpoints configured for this object adapter are ignored. This way, the client can give these proxies to the server (via the router), and when the server sends a request using such a proxy, the request is directed to the router and then forwarded to the client's object adapter. An object adapter configured for bidirectional dispatch (without a router) uses `Endpoints` and `PublishedEndpoints` as usual to compute the published endpoints of the proxies it creates; usually you will leave `Endpoints` and `PublishedEndpoints` empty and the object adapter will create proxies with no endpoints at all.

See Callbacks through Glacier2 for an example.

Routing Tables

A router implementation may optionally maintain an internal routing table that the Ice client run time populates automatically by calling `addProxies`.

For example, Glacier2 uses a routing table because it can forward requests to any number of back-end servers, whereas IceBridge does not use a routing table because each IceBridge instance is statically configured to forward requests to a single server. When a client makes an initial invocation on a routed proxy, and the configured router uses a routing table, the Ice run time in the client needs to send that routed proxy to the router so that the router has the endpoint information necessary to establish a connection to the back-end server. The Ice run time in the client also maintains its own local version of the routing table in order to minimize overhead; Ice only sends each routed proxy to the router once. Furthermore, Ice keeps its table synchronized with the router's by tracking any proxies that the router might have evicted from its table (the proxies returned by the call to `addProxies`).

> Ice uses object identities as the keys in its routing table, which means it's important that your Ice objects use unique identities.

If a router receives an invocation to be forwarded for an object identity that is not in its routing table (e.g., it may have been recently evicted), the router raises `Ice::ObjectNotExistException` and sets the `operation` member to the reserved value `ice_add_proxy` in order to notify the Ice run time in the client that the routed proxy is unknown. The Ice client run time must therefore register the proxy with the router and retry the invocation.

See Also

- Glacier2
- IceBridge

Slicing Values and Exceptions

This page describes the concept of *slicing*, which is how the Ice run time reacts when it receives an instance of an unknown Slice class or exception.

On this page:

Composition using Slices

Classes and exceptions are composed of slices, where each slice corresponds to a level in the type hierarchy and contains the data members defined at that level. Consider this example showing a class hierarchy:

Slice

```
class A
{
    int i;
}

class B extends A
{
    float f;
}

class C extends B
{
    string s;
}
```

An instance of class `C` contains the following slices, listed in order from most-derived to least-derived:

Slice	Contents
`C`	`string s`
`B`	`float f`
`A`	`int i`

Let's add the following interface to our discussion:

Slice

```
interface I
{
    B getObject();
}
```

Now suppose a client invokes `getObject`. Let's also suppose that the server actually returns an instance of class `C`, which is legal given

that C derives from B. The Ice run time in the client knows that the formal return type of the `getObject` operation is B, therefore B is the least-derived type that the client can accept for this operation. At a high level, the Ice run time in the client behaves as follows:

1. It discovers that the most-derived type of the returned instance is C and checks whether this type is known. These checks include language-specific mechanisms as well as the value factory API.
2. If this type is known to the client, Ice instantiates the object, extracts the data members for each of its slices, and returns the object.
3. If type C is *not* known to the client, which can occur when the client and server are using different versions of the Slice definitions, Ice discards the data members for slice C (also known as *slicing* the object) and tries again with the next slice.
4. If type B is also not known to the client then we have a problem. First, from a logical standpoint, the client *must* know type B because it is the statically-declared return type of the operation that the client just invoked. Second, we cannot slice this object any further because it would no longer be compatible with the formal signature of the operation; the returned object must at least be an instance of B, so we could not return an instance of A. In either case, the Ice run time would raise an exception.

Generally speaking, upon receipt of an instance of a class or exception, the Ice run time discards the slices of unknown types until it finds a type that it recognizes, exhausts all slices, or can no longer satisfy the formal type signature of the operation. This slicing feature allows the receiver, whose Slice definitions may be limited or outdated, to continue to function properly even when it does not recognize the most-derived type.

Slice Formats

Ice provides two on-the-wire formats for class and exception slices: the compact format and the sliced format. Ice uses the compact format by default, which is more space-efficient on the wire but offers less flexibility on the receiving end.

An application that needs the slicing behavior we discussed in the previous section must explicitly enable the sliced format as follows:

- Set the `Ice.Default.SlicedFormat` property to a non-zero value to force the Ice run time to use the sliced format by default.
- Annotate your Slice definitions with `format` metadata to selectively enable the sliced format for certain operations or interfaces.

For example, suppose an application can safely use the compact format most of the time, but still needs slicing in a few situations. In this case the application can use metadata to enable the sliced format where necessary:

Slice

```
interface Ledger
{
    Account getAccount(string id); // Uses compact format

    ["format:sliced"]
    Account importAccount(string source); // Uses sliced format
}
```

The semantics implied by these definitions state that the caller of `getAccount` assumes it will know every type that that might be returned, but the same cannot be said for `importAccount`. By enabling the sliced format here, we allow the client to "slice off" what it does not recognize, even if that means the client is left with only an instance of `Account` and not an instance of some derived type.

Now let's examine the opposite case: use the sliced format by default, and the compact format only in certain cases:

Slice

```
["format:sliced"]
interface Ledger
{
    ["format:compact"]
    Account getAccount(string id); // Uses compact format

    Account importAccount(string source); // Uses sliced format
}
```

Here we specify that all operations in `Ledger` use the sliced format unless overridden at the operation level, which we do for `getAccount`.

The format affects the input parameters, output parameters, and exceptions of an operation. Consider this example:

Slice

```
exception IncompatibleAccount { ... }

interface Ledger
{
    ["format:compact"]
    Account migrateAccount(Account oldAccount) throws
IncompatibleAccount;
}
```

The metadata forces the client to use the compact format for the input parameter `oldAccount`, and forces the server to use the compact format for the return value or the exception. For a given operation, it is not possible to use one format for the parameters and a different format for the exceptions.

If you decide to use the `Ice.Default.SlicedFormat` property, be aware that this property only affects the sender of a value or exception. For example, if you enable this property in the client but not the server, then all values sent by the client use the sliced format by default, but all values and exceptions returned by the server use the compact format by default.

By offering two alternative formats, Ice gives you a great deal of flexibility in designing your applications. The compact format is ideal for applications that place a greater emphasis on efficiency, while the sliced format is helpful when clients and servers evolve independently.

Optional Objects

A limitation of the compact format can prevent a receiver from successfully receiving a message. Suppose a client uses the following Slice definitions:

Slice

```
class UserInfo
{
    string name;
    optional(1) string organization;
}
```

Also suppose that the server is using a newer version of these definitions:

Slice

```
class GroupInfo // New type
{
    ...;
}

class UserInfo
{
    string name;
    optional(1) string organization;
    optional(2) GroupInfo group; // New member
}
```

The server's definitions introduce a new class, `GroupInfo`, and an optional data member that uses this new type.

Let's examine the case where a client receives a message containing a `UserInfo` instance whose `group` member is set to a non-nil value. Since the `GroupInfo` type is unknown to the client, it needs to be able to skip the data associated with the `GroupInfo` instance in order to process the rest of the message. The sliced format includes the information the client needs to skip instances of unknown types, but the compact format unfortunately does not include this information and the client will raise an exception.

The lesson here is that adding a new class type to your application can have unexpected repercussions, especially when using the compact format. Remember that the *sender* can change formats without necessarily needing to update the receiver. For example, after adding the `group` member to `UserInfo`, the server could begin using the sliced format for all operations that return `UserInfo` to ensure that any existing clients would be able to receive it successfully.

Preserving Slices

The concept of slicing involves discarding the slices of unknown types when receiving an instance of a Slice class or exception. Here is a simple example:

Slice

```
class Base
{
    int b;
}

class Intermediate extends Base
{
    int i;
}

class Derived extends Intermediate
{
    int d;
}

interface Relay
{
    Base transform(Base b);
}
```

The server implementing the `Relay` interface must know the type `Base` (because it is statically referenced in the interface definition), but may not know `Intermediate` or `Derived`. Suppose the implementation of `transform` involves forwarding the instance to another back-end server for processing and returning the transformed instance to the caller. In effect, the `Relay` server is an intermediary. If the `Relay` server does not know the types `Intermediate` and `Derived`, it will slice an instance to `Base` and discard the data members of any more-derived types, which is clearly not the intended result because the back-end server *does* know those types. The only way the `Relay` server could successfully forward these instances is by knowing all possible derived types, which makes the application more difficult to evolve over time because the intermediary must be updated each time a new derived type is added.

To address this limitation, Ice supports a new metadata directive that allows an instance of a Slice class or exception to be forwarded with all of its slices intact, even if the intermediary does not know one or more of the instance's derived types. The new directive, `preserve-slice`, is shown below:

Slice

```
["preserve-slice"]
class Base
{
    int b;
}

class Intermediate extends Base
{
    int i;
}

class Derived extends Intermediate
{
    int d;
}

["format:sliced"]
interface Relay
{
    Base transform(Base b);
}
```

With this change, all instances of `Base`, and types derived from `Base`, can be forwarded intact. Also notice the addition of the `format:sliced` metadata on the `Relay` interface, which ensures that its operations use the sliced format and not the default compact format.

If a preserved value instance is sliced upon receipt, calling `ice_getSlicedData` on the value will return a `SlicedData` object that represents the complete state of the instance. The `SlicedData` class is described below:

C++11C++98C#JavaJava CompatJavaScriptMATLABObjective-CPHPPythonRuby

```
namespace Ice
{
    struct SliceInfo
    {
        std::string typeId;
        int compactId;
        std::vector<Byte> bytes;
        std::vector<std::shared_ptr<Value>> instances;
        bool hasOptionalMembers;
        bool isLastSlice;
    };

    class SlicedData
    {
    public:
        SlicedData(const SliceInfoSeq&);
        const SliceInfoSeq slices;
        void clear();
    };
}
```

```
namespace Ice
{
    struct SliceInfo : public IceUtil::Shared
    {
        std::string typeId;
        int compactId;
        std::vector<Byte> bytes;
        std::vector<ValuePtr> instances;
        bool hasOptionalMembers;
        bool isLastSlice;
    };

    class SlicedData : public IceUtil::Shared
    {
    public:
        SlicedData(const SliceInfoSeq&);
        const SliceInfoSeq slices;
        void clear();
    };
}
```

C#

```
namespace Ice
{
    public class SliceInfo
    {
        public string typeId;
        public int compactId;
        public byte[] bytes;
        public Value[] instances;
        public bool hasOptionalMembers;
        public bool isLastSlice;
    }

    public class SlicedData
    {
        public SlicedData(SliceInfo[] slices);

        public SliceInfo[] slices;
    }
}
```

Java

```
package com.zeroc.Ice;

public class SliceInfo
{
    public String typeId;
    public int compactId;
    public byte[] bytes;
    public com.zeroc.Ice.Value[] instances;
    public boolean hasOptionalMembers;
    public boolean isLastSlice;
}

public class SlicedData
{
    public SlicedData(SliceInfo[] slices);

    public SliceInfo[] slices;
}
```

Java

```
package Ice;

public class SliceInfo
{
    public String typeId;
    public int compactId;
    public byte[] bytes;
    public Ice.Object[] instances;
    public boolean hasOptionalMembers;
    public boolean isLastSlice;
}

public class SlicedData
{
    public SlicedData(SliceInfo[] slices);

    public SliceInfo[] slices;
}
```

JavaScript

```
class SliceInfo
{
    constructor()
    {
        this.typeId = "";
        this.compactId = -1;
        this.bytes = [];
        this.instances = [];
        this.hasOptionalMembers = false;
        this.isLastSlice = false;
    }
}

class SlicedData
{
    constructor(slices)
    {
        this.slices = slices;
    }
}
```

MATLAB

```
classdef SliceInfo < handle
    properties
        typeId
        compactId
        bytes
        instances
        hasOptionalMembers
        isLastSlice
    end
end

classdef SlicedData < handle
    properties(SetAccess=private)
        slices
    end
    methods
        function obj = SlicedData(slices)
            ...
        end
    end
end
```

Objective-C

```
ICE_API @protocol ICESlicedData<NSObject>
//
// Clear the slices to break potential cyclic references.
//
-(void) clear;
@end
```

PHP

```
class SliceInfo
{
    public $typeId;
    public $compactId;
    public $bytes;
    public $instances;
    public $hasOptionalMembers;
    public $isLastSlice;
}

class SlicedData
{
    public $slices;
}
```

Python

```
class SliceInfo(object):
    def __init__(self):
        self.typeId = ""
        self.compactId = -1
        self.bytes = None
        self.instances = None
        self.hasOptionalMembers = False
        self.isLastSlice = False

class SlicedData(object):
    def __init__(self, slices):
        self.slices = slices
```

Ruby

```
class SliceInfo
    attr_accessor :typeId, :compactId, :bytes, :instances,
:hasOptionalMembers, :isLastSlice
end

class SlicedData
    attr_accessor :slices   # array of SliceInfo
end
```

Applications do not normally need to use these types, but they provide all of the information the run time requires to forward an instance.

In C++11 and Objective-C, if the received value refers to a graph with cycles, the application should call `clear` on the `SlicedDat`

a object associated with the value to break potential cycles.

Unknown Sliced Values

Suppose we modify our `Relay` example as shown below:

Slice

```
["preserve-slice"]
class Base
{
    int b;
}

class Intermediate extends Base
{
    int i;
}

class Derived extends Intermediate
{
    int d;
}

["format:sliced"]
interface Relay
{
    Value transform(Value b);
}
```

The only difference here is the signature of the `transform` operation, which now uses the `Value` type. Technically, it is not necessary for the intermediary server to know *any* of the class types that might be relayed via this new definition of `transform` because the formal types in its signature do not impose any requirements. As long as any value types known by the intermediary are marked with `preserve-slice`, and the `transform` operation uses the sliced format, this intermediary is capable of relaying values of any type.

If the Ice run time in the intermediary does not know any of the types in an object's inheritance hierarchy, and the formal type is `Value`, Ice uses an instance of `UnknownSlicedValue` to represent the instance:

C++11C++98C#JavaJava CompatJavaScriptMATLABObjective-CPHPPythonRuby

```
namespace Ice
{
    class UnknownSlicedValue : public Value
    {
    public:

        std::string ice_id() const;
        std::shared_ptr<SlicedData> ice_getSlicedData() const;

        ...
    };
}
```

```
namespace Ice
{
    class UnknownSlicedValue : public Object
    {
    public:

        const std::string& ice_id() const;
        SlicedDataPtr ice_getSlicedData() const;

        ...
    };
}
```

C#

```
namespace Ice
{
    public sealed class UnknownSlicedValue : Value
    {
        public UnknownSlicedValue(string unknownTypeId);
        public override SlicedData ice_getSlicedData();
        public override string ice_id();
    }
}
```

Java

```
package com.zeroc.Ice;

public final class UnknownSlicedValue extends Value
{
    public UnknownSlicedValue(String unknownTypeId);
    public SlicedData ice_getSlicedData();
    public String ice_id();
}
```

Java

```
package Ice;

public final class UnknownSlicedValue extends ObjectImpl
{
    public UnknownSlicedValue(String unknownTypeId);
    public SlicedData ice_getSlicedData();
    public String ice_id();
}
```

JavaScript

```
class UnknownSlicedValue extends Ice.Value
{
    constructor(unknownTypeId)
    {
        ...
    }

    ice_getSlicedData()
    {
        ...
    }

    ice_id()
    {
        ...
    }
}
```

MATLAB

```
classdef UnknownSlicedValue < Ice.Value
    methods
        function obj = UnknownSlicedValue(unknownTypeId)
            ...
        end
        function r = ice_getSlicedData(obj)
            ...
        end
        function id = ice_id(obj)
            ...
        end
    end
end
```

Objective-C

```
ICE_API @interface ICEUnknownSlicedValue : ICEObject
{
@private
    NSString* unknownTypeId_;
    id<ICESlicedData> slicedData_;
}
@end
```

PHP

```
class UnknownSlicedValue extends Value
{
    public function ice_id()
    {
        ...
    }

    public function ice_getSlicedData()
    {
        ...
    }
}
```

Python

```
class UnknownSlicedValue(Value):
    def ice_id(self):
        ...
    def ice_getSlicedData(self):
        ...
```

Ruby

```
class UnknownSlicedValue < Value
    def ice_id
        ...
    end
    def ice_getSlicedData
        ...
    end
end
```

The implementation of `transform` receives an instance of `UnknownSlicedValue` and can use that object as its return value. If necessary, the implementation can determine the most-derived type of the instance by calling `ice_id`.

See Also

- Classes
- User Exceptions
- Type IDs
- Value Factories
- Slice Metadata Directives

Dynamic Ice

The Ice streaming API allows you to serialize and deserialize Slice types using the Ice encoding. This is useful, for example, if you want to store Slice types in a database.

The dynamic invocation and dispatch interfaces allow you to write generic clients and servers that need not have compile-time knowledge of the Slice types used by an application. This makes it possible to create applications such as object browsers, protocol analyzers, or protocol bridges. In addition, the dynamic invocation and dispatch interfaces permit services such as IceStorm to be implemented without the need to unmarshal and remarshal every message, with considerable performance improvements.

Keep in mind that applications that use dynamic invocation and dispatch are tedious to implement and harder to prove correct (because what normally would be a compile-time error appears only as a run-time error with dynamic invocation and dispatch). Therefore, you should use the dynamic interfaces only if your application truly benefits from this trade-off.

Topics

Streaming Interfaces

Ice provides convenient interfaces for streaming Slice types to and from a sequence of bytes. You can use these interfaces in many situations, such as when serializing types for persistent storage, and when using Ice's dynamic invocation and dispatch facility.

The streaming interfaces are not defined in Slice, but are rather a collection of native classes provided by each language mapping.

> The streaming interfaces are currently supported in C++, Java, JavaScript and .NET.

There are two primary abstract classes: `InputStream` and `OutputStream`. As you might guess, `InputStream` is used to extract Slice types from a sequence of bytes, while `OutputStream` is used to convert Slice types into a sequence of bytes. The classes provide the functions necessary to manipulate all of the core Slice types:

- Primitives (`bool`, `int`, `string`, etc.)
- Sequences of primitives
- Proxies
- Objects

The classes also provide functions that handle various details of the Ice encoding. Using these functions, you can manually insert and extract constructed types, such as dictionaries and structures, but doing so is tedious and error-prone. To make insertion and extraction of constructed types easier, the Slice compilers generate type-specific code that manages the low-level details for you.

The remainder of this section describes the streaming interfaces for each supported language mapping. To properly use the streaming interfaces, you should be familiar with the Ice encoding. For an example that demonstrates the use of the streaming interfaces, refer to the `Ice/invoke` directory in the Ice demo distribution.

Topics

See Also

- Dynamic Invocation and Dispatch
- Data Encoding

C++ Streaming Interfaces

The stream API allows you to manually marshal and unmarshal Slice types using the Ice data encoding.

Topics

The InputStream Interface in C++

On this page:

Initializing an `InputStream` in C++

The `InputStream` class provides a number of overloaded constructors:

C++11 C++98

```
namespace Ice
{
    class InputStream
    {
    public:
        InputStream();
        InputStream(const std::vector<Byte>&);
        InputStream(const std::pair<const Byte*, const Byte*>&);

        InputStream(const std::shared_ptr<Communicator>&);
        InputStream(const std::shared_ptr<Communicator>&, const
std::vector<Byte>&);
        InputStream(const std::shared_ptr<Communicator>&, const
std::pair<const Byte*, const Byte*>&);

        InputStream(const EncodingVersion&);
        InputStream(const EncodingVersion&, const std::vector<Byte>&);
        InputStream(const EncodingVersion&, const std::pair<const Byte*,
const Byte*>&);

        InputStream(const std::shared_ptr<Communicator>&, const
EncodingVersion&);
        InputStream(const std::shared_ptr<Communicator>&, const
EncodingVersion&, const std::vector<Byte>&);
        InputStream(const std::shared_ptr<Communicator>&, const
EncodingVersion&, const std::pair<const Byte*, const Byte*>&);

        ...
    };
}
```

```
namespace Ice
{
    class InputStream
    {
    public:
        InputStream();
        InputStream(const std::vector<Byte>&);
        InputStream(const std::pair<const Byte*, const Byte*>&);

        InputStream(const CommunicatorPtr&);
        InputStream(const CommunicatorPtr&, const std::vector<Byte>&);
        InputStream(const CommunicatorPtr&, const std::pair<const Byte*,
const Byte*>&);

        InputStream(const EncodingVersion&);
        InputStream(const EncodingVersion&, const std::vector<Byte>&);
        InputStream(const EncodingVersion&, const std::pair<const Byte*,
const Byte*>&);

        InputStream(const CommunicatorPtr&, const EncodingVersion&);
        InputStream(const CommunicatorPtr&, const EncodingVersion&,
const std::vector<Byte>&);
        InputStream(const CommunicatorPtr&, const EncodingVersion&,
const std::pair<const Byte*, const Byte*>&);

        ...
    };
}
```

The constructors accept three types of arguments:

- A communicator instance
- An encoding version
- The encoded data that you intend to decode

You'll normally supply the encoded data argument, which the stream accepts as either a vector or a pair of pointers to `Ice::Byte`.

> For efficiency reasons, the `InputStream` does not copy the data argument - it only references it.

We recommend supplying a communicator instance, otherwise you will not be able to decode proxy objects. The stream also inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using functions that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` functions:

C++11 C++98

```
class InputStream
{
public:

   void initialize(const std::shared_ptr<Communicator>&);
   void initialize(const std::shared_ptr<Communicator>&, const
EncodingVersion&);
   ...
};
```

```
class InputStream
{
public:
   void initialize(const CommunicatorPtr&);
   void initialize(const CommunicatorPtr&, const EncodingVersion&);
   ...
};
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following functions to manually configure the stream:

C++11 C++98

```
namespace Ice
{
    class InputStream
    {
    public:
        void setValueFactoryManager(const
std::shared_ptr<ValueFactoryManager>&);
        void setLogger(const std::shared_ptr<Logger>&);
        void setCompactIdResolver(std::function<std::string (int)>);
        void setSliceValues(bool);
        void setTraceSlicing(bool);

        ...
    };
}
```

C++98

```
namespace Ice
{
    class InputStream
    {
    public:
        void setValueFactoryManager(const ValueFactoryManagerPtr&);
        void setLogger(const LoggerPtr&);
        void setCompactIdResolver(const CompactIdResolverPtr&);
        void setCollectObjects(bool);
        void setSliceValues(bool);
        void setTraceSlicing(bool);

        ...
    };
}
```

The settings include:

- Value factory manager
  A value factory manager supplies custom factories for Slice class types.
- Logger
  The stream uses a logger to record warning and trace messages.
- Compact ID resolver
  A compact ID resolver for translating numeric values into Slice type IDs. The stream invokes the resolver by passing it the numeric compact ID. The resolver is expected to return the Slice type ID associated with that numeric ID or an empty string if the numeric ID is unknown. The C++11 API accepts a `std::function` whereas the C++98 API requires an instance of `Ice::CompactIdResolver`:

  C++98

  ```
  class CompactIdResolver : public IceUtil::Shared
  {
  public:
      virtual std::string resolve(Ice::Int) const = 0;
  };
  typedef IceUtil::Handle<CompactIdResolver> CompactIdResolverPtr;
  ```

- Collectable
  The flag indicates whether to mark instances of Slice classes as collectable (C++98 only). If the stream is initialized with a communicator, this setting defaults to the value of the `Ice.CollectObjects` property, otherwise the setting defaults to false.
- Slice values
  The flag indicates whether to slice instances of Slice classes to a known Slice type when a more derived type is unknown. An instance is "sliced" when no static information is available for a Slice type ID and no factory can be found for that type, resulting in the creation of an instance of a less-derived type. If slicing is disabled in this situation, the stream raises the exception `NoValueFactoryException`. The default behavior is to allow slicing.
- Trace slicing
  The flag indicates whether to log messages when instances of Slice classes are sliced. If the stream is initialized with a communicator, this setting defaults to the value of the `Ice.Trace.Slicing` property, otherwise the setting defaults to false.

Extracting from an `InputStream` in C++

`InputStream` provides a number of overloaded read member functions that allow you to extract nearly all Slice types from the stream simply by calling `read`.

For example, you can extract a double value followed by a string from a stream as follows:

C++

```
vector<Ice::Byte> data = ...;
Ice::InputStream in(communicator, data);
double d;
in.read(d);
string s;
in.read(s);
```

Likewise, you can extract a sequence of a built-in type, or a complex type, from the stream as follows:

C++

```
vector<Ice::Byte> data = ...;
Ice::InputStream in(communicator, data);
// ...
IntSeq s; // Slice: sequence<int> IntSeq;
in.read(s);

ComplexType c;
in.read(c);
```

With the C++11 mapping, you can also call `readAll` with all the parameters to extract, for example the sample above can be rewritten as:

C++11

```
vector<Ice::Byte> data = ...;
Ice::InputStream in(communicator, data);
// ...
IntSeq s; // Slice: sequence<int> IntSeq;
ComplexType c;
in.readAll(s, c);
```

Here are most of the functions for extracting data from a stream:

C++11 C++98

```
class InputStream
{
public:

    void read(bool&);
    void read(Byte&);
```

```
    void read(short&);
    void read(int&);
    void read(long long int&);
    void read(float&);
    void read(double&);

    void read(std::string& s, bool convert = true);
    void read(std::vector<std::string>& v, bool convert = true);
    void read(std::wstring&);
    void read(std::vector<std::wstring>&);

    void read(PatchFunc patchFunc, void* patchAddr);

    int readEnum(int maxValue);

    void throwException(std::function<void(const std::string&)> =
nullptr);

    void readBlob(std::vector<Byte> v, int sz);
    void readBlob(const Byte*& v, size_type sz);

    template<typename T> inline void read(T& v);

    void read(std::pair<const Byte*, const Byte*>&);
    void read(std::vector<bool>&);
    void read(std::vector<Byte>&);
    void read(std::vector<short>&);
    void read(std::vector<int>&);
    void read(std::vector<long long int>&);
    void read(std::vector<float>&);
    void read(std::vector<double>&);

    // "Zero-copy" read
    void read(const char*& vdata, size_t& vsize, bool convert = true);
    void read(std::pair<const bool*, const bool*>&);
    void read(std::pair<const short*, const short*>&);
    void read(std::pair<const int*, const int*>&);
    void read(std::pair<const long long int*, const long long int*>&);
    void read(std::pair<const float*, const float*>&);
    void read(std::pair<const double*, const double*>&);

    // Extract a proxy of the base interface type
    std::shared_ptr<ObjectPrx> readProxy();

    // Extract a proxy of a user-defined interface type
    template</* T is-a ObjectPrx */> void read(std::shared_ptr<T>& v);

    // Extract an instance of a Slice class
    template</* T is-a Value */> void read(::std::shared_ptr<T>& v);
```

```
    void startValue();
    std::shared_ptr<SlicedData> endValue(bool preserve);

    void startException();
    std::shared_ptr<SlicedData> endException(bool preserve);

    // Read a list of "mandatory" parameters or data members in a single
call
    template<typename T> void readAll(T& v);
    template<typename T, typename... Te> void readAll(T& v, Te&... ve);

    // Read a list of optional parameters or data members in a single
call
    template<typename T> void readAll(std::initializer_list<int> tags,
Optional<T>& v);
    template<typename T, typename... Te> void
readAll(std::initializer_list<int> tags, Optional<T>& v,
Optional<Te>&... ve);

    int readSize();
    int readAndCheckSeqSize(int minWireSize);

    std::string startSlice();
    void endSlice();
    void skipSlice();

    EncodingVersion startEncapsulation();
    void endEncapsulation();
    EncodingVersion skipEncapsulation();
    EncodingVersion skipEmptyEncapsulation();
    int getEncapsulationSize();

    EncodingVersion getEncoding();

    void readPendingValues();

    size_type pos();
    void pos(size_type p);

    void skip(int sz);
    void skipSize();

    bool readOptional(int tag, OptionalFormat expectedFormat);
    void skipOptional(OptionalFormat format);
    void skipOptionals();
```

```
    template<typename T> void read(int tag, Ice::Optional<T>& v)
};
```

```
class InputStream
{
public:

    void read(bool&);
    void read(Byte&);
    void read(short&);
    void read(int&);
    void read(long long int&);
    void read(float&);
    void read(double&);

    void read(std::string& s, bool convert = true);
    void read(std::vector<std::string>& v, bool convert = true);
    void read(std::wstring&);
    void read(std::vector<std::wstring>&);

    void read(PatchFunc patchFunc, void* patchAddr);

    int readEnum(int maxValue);

    void throwException(const Ice::UserExceptionFactorPtr& = 0);

    void readBlob(std::vector<Byte> v, int sz);
    void readBlob(const Byte*& v, size_type sz);

    template<typename T> inline void read(T& v);

    void read(std::pair<const Byte*, const Byte*>&);
    void read(std::vector<bool>&);
    void read(std::vector<Byte>&);
    void read(std::vector<short>&);
    void read(std::vector<int>&);
    void read(std::vector<long long int>&);
    void read(std::vector<float>&);
    void read(std::vector<double>&);

    // "Zero-copy" read
    void read(const char*& vdata, size_t& vsize); // does not use string
converter
    void read(const char*& vdata, size_t& vsize, std::string& holder);
// uses string converter
    void read(std::pair<const bool*, const bool*>& v,
IceUtil::ScopedArray<bool>& holder);
```

```
    void read(std::pair<const Short*, const Short*>& v,
IceUtil::ScopedArray<Short>& holder);
    void read(std::pair<const Int*, const Int*>& v,
IceUtil::ScopedArray<Int>& holder);
    void read(std::pair<const Long*, const Long*>& v,
IceUtil::ScopedArray<Long>& holder);
    void read(std::pair<const Float*, const Float*>& v,
IceUtil::ScopedArray<Float>& holder);
    void read(std::pair<const Double*, const Double*>& v,
IceUtil::ScopedArray<Double>& holder);
    // Extract a proxy of the base interface type
    void read(ObjectPrx&);

    // Extract a proxy of a user-defined interface type
    template<typename T> void read(ProxyHandle<T>& v);

    // Extract an instance of a Slice class
    template<typename T> void read(Handle<T>& v);

    void startValue();
    SlicedDataPtr endValue(bool preserve);

    void startException();
    SlicedDataPtr endException(bool preserve);

    int readSize();
    int readAndCheckSeqSize(int minWireSize);

    std::string startSlice();
    void endSlice();
    void skipSlice();

    EncodingVersion startEncapsulation();
    void endEncapsulation();
    EncodingVersion skipEncapsulation();
    EncodingVersion skipEmptyEncapsulation();
    int getEncapsulationSize();

    EncodingVersion getEncoding();

    void readPendingValues();

    size_type pos();
    void pos(size_type p);

    void skip(int sz);
    void skipSize();

    bool readOptional(int tag, OptionalFormat expectedFormat);
    void skipOptional(OptionalFormat format);
```

```
void skipOptionals();
```

```
    template<typename T> void read(int tag, Ice::Optional<T>& v)
};
```

Some of these functions need more explanation:

- `void read(std::string& s, bool convert = true)`
  The bool argument determines whether the strings unmarshaled by `read` are processed by the string converter, if one is installed. The default behavior is to convert the strings.

- `int readEnum(int maxValue)`
  Unmarshals the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.
  In general, you should simply use read for your enum values. read with an enum parameter calls `readEnum` with the `maxValue` provided by the code generated by `slice2cpp`.

- `int readSize()`
  The Ice encoding has a compact representation to indicate size. This function extracts a size and returns it as an integer.

- `int readAndCheckSeqSize(int minWireSize)`
  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element. If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `std::shared_ptr<ObjectPrx> readProxy() (C++11)`
  `void read(ObjectPrx&) (C++98)`
  Returns an instance of the base proxy type, `ObjectPrx`. Calling `read` with a proxy parameter has the same effect.

- `void read(std::shared_ptr<T>& v) (C++11)`
  `void read(Handle<T>& v) (C++98)`
  `void read(PatchFunc patchFunc, void* patchAddr)`

  These functions are used to extract instances of Slice classes. The Ice encoding for class instances requires extraction to occur in stages. The final overloading above accepts a function pointer, whose definition is shown below:
  C++11 C++98

  ```
  typedef void (*PatchFunc)(void*, const std::shared_ptr<Value>&);
  ```

  ```
  typedef void (*PatchFunc)(void*, const ValuePtr&);
  ```

  When the instance is available, the stream invokes the function and passes the value of `patchAddr` as well as a smart pointer for the new instance. The application must call `readPendingValues` to ensure that all instances are properly extracted. If you're not interested in receiving a callback when the instance is extracted, it is easier to call the `read` overload that accepts a smart pointer parameter. Note that calling `endEncapsulation` implicitly calls `readPendingValues` if necessary.

- `void throwException(std::function<void(const std::string&)> = nullptr) (C++11)`
  `void throwException(const UserExceptionFactoryPtr& factory = 0) (C++98)`

This function extracts a user exception from the stream and throws it. If the stored exception is of an unknown type, the function attempts to extract and throw a less-derived exception. If that also fails, an exception is thrown: for the 1.0 encoding, the exception is `UnmarshalOutOfBoundsException`, for the 1.1 encoding, the exception is `UnknownUserException`. You can optionally supply a factory function in C++11, or an object that implements the `Ice::UserExceptionFactory` bstract base class in C++98:

C++98

```
class UserExceptionFactory : public IceUtil::Shared
{
public:
    virtual void createAndThrow(const std::string& typeId) const =
0;
};
typedef ... UserExceptionFactoryPtr;
```

As the stream iterates over the slices of an exception from most-derived to least-derived, it invokes `createAndThrow` passing the type ID of each slice, giving the application an opportunity to raise an instance of `UserException`. The stream expects the exception instance to call `startException`, unmarshal the remaining slices, and then call `endException`. If the factory does not throw an exception for a type ID, and no static information can be found for the type, the stream skips that slice.

- `void startValue()`
  `std::shared_ptr<SlicedData> endValue(bool preserve) (C++11)`
  `SlicedDataPtr endValue(bool preserve) (C++98)`
  The `startValue` function must be called prior to reading the slices of a class instance. The `endValue` function must be called after all slices have been read. Pass true to `endValue` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endValue` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the value with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `void startException()`
  `std::shared_ptr<SlicedData> endException(bool preserve) (C++11)`
  `SlicedDataPtr endException(bool preserve) (C++98)`
  The `startException` function must be called prior to reading the slices of an exception. The `endException` function must be called after all slices have been read. Pass true to `endException` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endException` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the exception with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `std::string startSlice()`
  `void endSlice()`
  `void skipSlice()`
  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of a class instance or user exception. The `startSlice` function returns the type ID of the next slice, which may be an empty string depending on the format used to encode the instance or exception.

- `EncodingVersion startEncapsulation()`
  `void endEncapsulation()`
  `EncodingVersion skipEncapsulation()`
  Start, end, and skip an encapsulation, respectively. The `startEncapsulation` and `skipEncapsulation` functions return the encoding version used to encode the contents of the encapsulation.

- `EncodingVersion skipEmptyEncapsulation()`
  Skips an encapsulation that is expected to be empty. The stream raises `EncapsulationException` if the encapsulation is not empty.

- `int getEncapsulationSize()`
  Returns the size of the current encapsulation.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently in use by the stream.

- `void readPendingValues()`
  An application must call this function after all other data has been extracted. This function extracts the state of class instances and

invokes their corresponding callback objects. For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `size_type pos()`
  `void pos(size_type p)`
  Queries or modifies the current position of the stream.

- `void skip(int sz)`
  Skips the given number of bytes.

- `void skipSize()`
  Reads a size at the current position and skips that number of bytes.

- `bool readOptional(int tag, OptionalFormat fmt)`
  Returns true if an optional value with the given tag and format is present, or false otherwise. If this function returns true, the data associated with that optional value must be read next. Optional values must be read in order by tag from least to greatest. The `Ice::OptionalFormat` enumeration is defined as follows:

  C++11 C++98

```
enum class OptionalFormat : unsigned char
{
    F1 = 0,             // Fixed 1-byte encoding
    F2 = 1,             // Fixed 2 bytes encoding
    F4 = 2,             // Fixed 4 bytes encoding
    F8 = 3,             // Fixed 8 bytes encoding
    Size = 4,           // "Size encoding" on 1 to 5 bytes, e.g.
enum, class identifier
    VSize = 5,          // "Size encoding" on 1 to 5 bytes followed
by data, e.g. string, fixed size
                        // struct, or containers whose size can be
computed prior to marshaling
    FSize = 6,          // Fixed size on 4 bytes followed by data,
e.g. variable-size struct, container.
    Class = 7
};
```

```
enum OptionalFormat
{
    OptionalFormatF1 = 0, // See C++11
    OptionalFormatF2 = 1,
    OptionalFormatF4 = 2,
    OptionalFormatF8 = 3,
    OptionalFormatSize = 4,
    OptionalFormatVSize = 5,
    OptionalFormatFSize = 6,
    OptionalFormatClass = 7
};
```

  Refer to the encoding discussion for more information on the meaning of these values.

- `void read(int tag, Ice::Optional<T>& v) (C++11)`
  `void read(int tag, IceUtil::Optional<T>& v) (C++98)`
  If an optional value with the given tag is present, the `Optional` parameter is marked as set and its value is read.

- `void skipOptional(OptionalFormat format)`
  `void skipOptionals()`
  Skips one optional value with the given format, or skips all remaining optional values, respectively.
- `void setClosure(void* p)`
  `void* getClosure() const`
  Allows an arbitrary value to be associated with the stream.

Extracting Sequences of Built-In Types using Zero-Copy in C++

`InputStream` provides a number of overloads that accept a pair of pointers. For example, you can extract a sequence of bytes as follows:

C++

```
vector<Ice::Byte> data = ...;
Ice::InputStream in(communicator, data);
std::pair<const Ice::Byte*, const Ice::Byte*> p;
in.read(p);
```

The same extraction works for the other built-in integral and floating-point types, such `int` and `double`.

If the extraction is for a byte sequence, the returned pointers always point at memory in the stream's internal marshaling buffer.

For the other built-in types, the pointers refer to the internal marshaling buffer only if the Ice encoding is compatible with the machine and compiler representation of the type, otherwise the pointers refer to a temporary array allocated to hold the unmarshaled data. With the C++11 mapping, the `InputStream` itself allocates and holds these temporary arrays, whereas with the C++98, you need to supply this temporary array as a `IceUtil::ScopedArray` parameter.

Here is an example to illustrate how to extract a sequence of integers, regardless of whether the machine's encoding of integers matches the on-the-wire representation or not:

C++11 C++98

```
...
Ice::InputStream in(communicator, data);
std::pair<const int*, const int*> p;
in.read(p);

for(const int* i = p.first; i != p.second; ++i)
{
    cout << *i << endl;
}
```

```
...
Ice::InputStream in(communicator, data);
std::pair<const int*, const int*> p;
IceUtil::ScopedArray<int> holder;
in.read(p, holder);

for(const int* i = p.first; i != p.second; ++i)
{
    cout << *i << endl;
}
```

If the on-the-wire encoding matches that of the machine, and therefore zero-copy is possible, the returned pair of pointers points into the stream's internal marshaling buffer. Otherwise, the stream allocates an array, unmarshals the data into the array, and sets the pair of pointers to point into that array.

See Also

- Smart Pointers for Classes
- slice2cpp Command-Line Options (C++98)
- C++98 Strings and Character Encoding
- Data Encoding for Classes
- Basic Data Encoding
- The C++ ScopedArray Template

The OutputStream Interface in C++

On this page:

Initializing an `OutputStream` in C++

The `OutputStream` class provides a number of overloaded constructors:

`C++11` `C++98`

```
namespace Ice
{
    class OutputStream
    {
    public:
        OutputStream();

        OutputStream(const std::shared_ptr<Communicator>& communicator);

        OutputStream(const std::shared_ptr<Communicator>& communicator,
const EncodingVersion& version);

        OutputStream(const std::shared_ptr<Communicator>& communicator,
const EncodingVersion& version,
                const std::pair<const Byte*, const Byte*>& buf);
    };
}
```

```
namespace Ice
{
    class OutputStream
    {
    public:
        OutputStream();

        OutputStream(const CommunicatorPtr& communicator);

        OutputStream(const CommunicatorPtr& communicator, const
EncodingVersion& version);

        OutputStream(const CommunicatorPtr& communicator, const
EncodingVersion& version,
                    const std::pair<const Byte*, const Byte*>& buf);
    };
}
```

The constructors optionally accept the following arguments:

- A communicator instance
- An encoding version
- A pair of bytes denoting the beginning and end of a memory block to be used as the stream's initial marshaling buffer. This is useful to avoid memory allocations when the size of the encoded data is predictable. The stream will automatically allocate a larger buffer if the encoded data exceeds the size of the given memory block.

We recommend supplying a communicator instance. The stream inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using functions that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

Instances of `OutputStream` can be allocated statically or dynamically.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` functions:

C++11 C++98

```
class OutputStream
{
public:
    void initialize(const std::shared_ptr<Communicator>& communicator);
    void initialize(const std::shared_ptr<Communicator>& communicator,
const EncodingVersion& version);
    ...
};
```

```
class OutputStream
{
public:
    void initialize(const CommunicatorPtr& communicator);
    void initialize(const CommunicatorPtr& communicator, const
EncodingVersion& version);
    ...
};
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following function to manually configure the stream:

C++

```
class OutputStream
{
public:
    void setFormat(FormatType);
    ...
};
```

For instances of Slice classes, the format determines how the slices of an instance are encoded. If the stream is initialized with a communicator, this setting defaults to the value of `Ice.Default.SlicedFormat`, otherwise the setting defaults to the compact format.

Inserting into an `OutputStream` in C++

`OutputStream` provides a number of overloaded `write` member functions that allow you to insert any parameter into the stream simply by calling `write`.

For example, you can insert a double value followed by a string into a stream as follows:

C++

```
Ice::OutputStream out(communicator);
Ice::Double d = 3.14;
out.write(d);
string s = "Hello";
out.write(s);
```

Likewise, you can insert a sequence of built-in type, or a complex, with the same syntax:

C++

```
Ice::OutputStream out(communicator);
IntSeq s = ...;
out.write(s);

ComplexType c = ...;
out.write(c);
```

With the C++11 mapping, you can also write a list of parameters in one call with `writeAll`. For example, the previous sample can be rewritten as:

C++11

```
Ice::OutputStream out(communicator);
IntSeq s = ...;
ComplexType c = ...;
out.writeAll(s, c);
```

Here are the functions for inserting data into an stream:

C++11 C++98

```
class OutputStream : ...
{
public:

    void write(bool);
    void write(Byte);
    void write(short);
    void write(int);
    void write(long long int);
    void write(float);
    void write(double);
    void write(const std::string&, bool = true);
```

```
    void write(const char*, size_t, bool = true);
    void write(const char*, bool = true);
    void write(const std::string*, const std::string*, bool = true);

    void write(const std::wstring&);
    void write(const std::wstring*, const std::wstring*);

    template<typename T> void write(const std::vector<T>& v);

    template<typename T> void write(const T* begin, const T* end);

    void writeBlob(const std::vector<Byte>&);
    void writeBlob(const Byte* v, size_type sz);

    void writeEnum(int v, int maxValue);

    bool writeOptional(int tag, OptionalFormat fmt);

    template<typename T> void write(int tag, const Optional<T>& v);

    template<typename T> void write(const T& v);

    // Insert a proxy of the base type
    void writeProxy(const std::shared_ptr<ObjectPrx>&);

    // Insert a proxy of a user-defined interface type
    template</* T is-a ObjectPrx */>
    void write(const std::shared_ptr<T>& v);

    // Insert an instance of a Slice class
    template</* T is-a Value */>
    void write(const std::shared_ptr<T>& v);

    void startValue(const std::shared_ptr<SlicedData>& sd);
    void endValue();

    void startException(const std::shared_ptr<SlicedData>& sd);
    void endException();

    // Write all parameters in one call
    template<typename T> void writeAll(const T& v);
    template<typename T, typename... Te> void writeAll(const T& v, const
Te&... ve);

    // Write all optional parameters in one call
    template<typename T> void writeAll(std::initializer_list<int> tags,
const Optional<T>& v);
    template<typename T, typename... Te> void
writeAll(std::initializer_list<int> tags, const Optional<T>& v, const
Optional<Te>&... ve);
```

```
    void writeSize(int sz);
    size_type startSize();
    void endSize(size_type pos);
    void rewriteSize(int v, iterator dest);
    void write(int v, iterator dest);
    void rewrite(int v, size_type pos);

    void writeException(const UserException& e);

    void startSlice(const std::string& typeId, int compactId, bool
last);
    void endSlice();

    void startEncapsulation(const EncodingVersion& v, FormatType fmt);
    void startEncapsulation();
    void endEncapsulation();
    void writeEmptyEncapsulation(const EncodingVersion& v);
    void writeEncapsulation(const Byte* v, int sz);

    EncodingVersion getEncoding() const;

    void writePendingValues();

    void finished(std::vector<Byte>& v);
    std::pair<const Byte*, const Byte*> finished();

    void resize(size_type sz);
```

```
    size_type pos();
    void pos(size_type n);
};
```

```
class OutputStream : ...
{
public:

    void write(bool);
    void write(Byte);
    void write(short);
    void write(int);
    void write(long long int);
    void write(float);
    void write(double);
    void write(const std::string&, bool = true);
    void write(const char*, size_t, bool = true);
    void write(const char*, bool = true);
    void write(const std::string*, const std::string*, bool = true);

    void write(const std::wstring&);
    void write(const std::wstring*, const std::wstring*);

    template<typename T> void write(const std::vector<T>& v);

    template<typename T> void write(const T* begin, const T* end);

    void writeBlob(const std::vector<Byte>&);
    void writeBlob(const Byte* v, size_type sz);

    void writeEnum(int v, int maxValue);

    bool writeOptional(int tag, OptionalFormat fmt);

    template<typename T> void write(int tag, const IceUtil::Optional<T>&
v);

    template<typename T> void write(const T& v);

    // Insert a proxy of the base interface type
    void write(const ObjectPrx&);

    // Insert a proxy of a user-defined interface type
    template<typename T> void
    write(const ProxyHandle<T>& v);

    // Insert an instance of the base class type
```

```
    void write(const ObjectPtr& v);

    // Insert an instance of a user-defined class type
    template<typename T> void
    write(const Handle<T>& v);

    void startValue(const SlicedDataPtr& sd);
    void endValue();

    void startException(const SlicedDataPtr& sd);
    void endException();

    void writeSize(int sz);
    size_type startSize();
    void endSize(size_type pos);
    void rewriteSize(int v, iterator dest);
    void write(int v, iterator dest);
    void rewrite(int v, size_type pos);

    void writeException(const UserException& e);

    void startSlice(const std::string& typeId, int compactId, bool
last);
    void endSlice();

    void startEncapsulation(const EncodingVersion& v, FormatType fmt);
    void startEncapsulation();
    void endEncapsulation();
    void writeEmptyEncapsulation(const EncodingVersion& v);
    void writeEncapsulation(const Byte* v, int sz);

    EncodingVersion getEncoding() const;

    void writePendingValues();

    void finished(std::vector<Byte>& v);
    std::pair<const Byte*, const Byte*> finished();

    void resize(size_type sz);
```

```
    size_type pos();
    void pos(size_type n);
};
```

Some of the `OutputStream` functions need more explanation:

- `void write(const std::string& v, bool convert = true)`
  `void write(const char* v, size_t vlen, bool convert = true)`
  `void write(const char* v, bool convert = true)`
  `void write(const std::string* beg, const std::string* end, bool convert = true)`
  The boolean argument determines whether the strings marshaled by these functions are processed by the narrow string converter, if one has been provided. The default behavior is to convert the strings.

- `template<typename T> void write(const T* begin, const T* end)`
  Marshals the given range as a sequence of type `T`.

- `void writeBlob(const std::vector<Byte>&)`
  `void writeBlob(const Byte* v, size_type sz)`
  Copies the specified blob of bytes to the stream without modification.

- `void writeEnum(int val, int maxValue)`
  Writes the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.
  In general, you should simply use `write` for your enum values. `write` with an enum parameter calls `writeEnum` with the `maxValue` provided by the code generated by `slice2cpp`.

- `bool writeOptional(int tag, OptionalFormat fmt)`
  Prepares the stream to write an optional value with the given tag and format. Returns true if the value should be written, or false otherwise. A return value of false indicates that the encoding version in use by the stream does not support optional values. If this function returns true, the data associated with that optional value must be written next. Optional values must be written in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  C++11 C++98

```
enum class OptionalFormat : unsigned char
{
    F1 = 0,             // Fixed 1-byte encoding
    F2 = 1,             // Fixed 2 bytes encoding
    F4 = 2,             // Fixed 4 bytes encoding
    F8 = 3,             // Fixed 8 bytes encoding
    Size = 4,           // "Size encoding" on 1 to 5 bytes, e.g.
enum, class identifier
    VSize = 5,          // "Size encoding" on 1 to 5 bytes followed
by data, e.g. string, fixed size
                        // struct, or containers whose size can be
computed prior to marshaling
    FSize = 6,          // Fixed size on 4 bytes followed by data,
e.g. variable-size struct, container.
    Class = 7
};
```

```
enum OptionalFormat
{
    OptionalFormatF1 = 0,  // see C++11
    OptionalFormatF2 = 1,
    OptionalFormatF4 = 2,
    OptionalFormatF8 = 3,
    OptionalFormatSize = 4,
    OptionalFormatVSize = 5,
    OptionalFormatFSize = 6,
    OptionalFormatClass = 7
};
```

Refer to the encoding discussion for more information on the meaning of these values.

- `template<typename T> void write(int tag, const Ice::Optional<T>& v) (C++11)`
  `template<typename T> void write(int tag, const IceUtil::Optional<T>& v) (C++98)`
  If the given optional value is set, this function writes the optional with its value.

- `void writeSize(int sz)`
  The Ice encoding has a compact representation to indicate size. This function converts the given non-negative integer into the proper encoded representation.

- `size_type startSize()`
  `void endSize(size_type pos)`
  The encoding for optional values uses a 32-bit integer to hold the size of variable-length types. Calling `startSize` writes a placeholder value for the size and returns the starting position of the size value; after writing the data, call `endSize` to patch the placeholder with the actual size at the given position.

- `void rewriteSize(int v, iterator dest)`
  Replaces a size value at the given destination in the stream. This function does not change the stream's current position.

- `void write(int v, iterator dest)`
  `void rewrite(int v, size_type pos)`
  Overwrite a 32-bit integer value at the given destination or position in the stream. These functions do not change the stream's current position.

- `void writeProxy(const std::shared_ptr<ObjectPrx>&) (C++11)`
  `template</* T is-a ObjectPrx */> void write(const std::shared_ptr<T>& v) (C++11)`
  `void write(const ObjectPrx&) (C++98)`
  `template<typename T> void write(const ProxyHandle<T>& v) (C++98)`
  Inserts a proxy into the stream.

- `template</* T is-a Value */> void write(const std::shared_ptr<T>& v) (C++11)`
  `void write(const ObjectPtr& v) (C++98)`
  `template<typename T> void write(const Handle<T>& v) (C++98)`
  Inserts an instance of a Slice class. The Ice encoding for class instances may cause the insertion of this instance to be delayed, in which case the stream retains a reference to the given instance and the stream does not insert its state it until `writePendingValues` is invoked on the stream.

- `void writeException(const Ice::UserException& ex)`
  Inserts a user exception. It is equivalent to calling `write` with a user exception.

- `void startValue(const std::shared_ptr<SlicedData>& sd) (C++11)`
  `void startValue(const SlicedDataPtr& sd) (C++98)`
  `void endValue()`
  When marshaling the slices of a class instance, the application must first call `startValue`, then marshal the slices, and finally call `endValue`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or 0 if there are no preserved slices.

- `void startException(const std::shared_ptr<SlicedData>& sd) (C++11)`
  `void startException(const SlicedDataPtr& sd) (C++98)`
  `void endException()`
  When marshaling the slices of an exception, the application must first call `startException`, then marshal the slices, and finally call `endException`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or 0 if there are no preserved slices.

- `void startSlice(const std::string& typeId, int compactId, bool last)`
  `void endSlice()`
  Starts and ends a slice of class or exception member data. The call to `startSlice` must include the type ID for the current slice, the corresponding compact ID for the type (if any), and a boolean indicating whether this is the last slice of the class instance or exception. The compact ID is only relevant for class instances; pass a negative value to indicate the encoding should use the string type ID.

- `void startEncapsulation(const EncodingVersion& v, FormatType fmt)`
  `void startEncapsulation()`
  `void endEncapsulation()`
  Starts and ends an encapsulation, respectively. The first overloading of `startEncapsulation` allows you to specify the encoding version as well as the format to use for any class instances and exceptions marshaled within this encapsulation.

- `void writeEmptyEncapsulation(const EncodingVersion& v)`
  Writes an encapsulation having the given encoding version with no encoded data.

- `void writeEncapsulation(const Byte* v, int sz)`
  Copies the bytes representing an encapsulation from the given address into the stream.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently being used by the stream.

- `void writePendingValues()`
  Encodes the state of class instances whose insertion was delayed during a previous call to `write`. This member function must only be called once. For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `void finished(std::vector<Byte>& data)`
  `std::pair<const Byte*, const Byte*> finished()`
  Indicates that marshaling is complete. This member function must only be called once. In the first overloading, the given byte sequence is filled with a copy of the encoded data. The second overloading avoids a copy by returning a pair of pointers to the stream's internal memory; these pointers are valid for the lifetime of the `OutputStream` object.

- `void resize(size_type sz)`
  Allocates space for `sz` more bytes in the buffer. The stream implementation internally uses this function prior to copying more data into the buffer.

- `size_type pos()`
  `void pos(size_type n)`

Returns or changes the stream's current position, respectively.

See Also

- Basic Data Encoding
- slice2cpp Command-Line Options (C++98)
- C++98 Strings and Character Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions

Java Streaming Interfaces

The stream API allows you to manually marshal and unmarshal Slice types using the Ice data encoding.

Topics

The InputStream Interface in Java

On this page:

Initializing an `InputStream` in Java

The `InputStream` class provides a number of overloaded constructors:

Java

```
package com.zeroc.Ice;

public class InputStream
{
    public InputStream();
    public InputStream(byte[] data);
    public InputStream(java.nio.ByteBuffer buf);
    public InputStream(Communicator communicator);
    public InputStream(Communicator communicator, byte[] data);
    public InputStream(Communicator communicator, java.nio.ByteBuffer
buf);
    public InputStream(EncodingVersion encoding);
    public InputStream(EncodingVersion encoding, byte[] data);
    public InputStream(EncodingVersion encoding, java.nio.ByteBuffer
buf);
    public InputStream(Communicator communicator, EncodingVersion
encoding);
    public InputStream(Communicator communicator, EncodingVersion
encoding, byte[] data);
    public InputStream(Communicator communicator, EncodingVersion
encoding, java.nio.ByteBuffer buf);

    ...
}
```

The constructors accept three types of arguments:

- A communicator instance
- An encoding version
- The encoded data that you intend to decode

You'll normally supply the encoded data argument, which the stream accepts as either an array of bytes or a `ByteBuffer`. In either case, the stream does not make a copy of the data; rather, it uses the data as supplied and assumes it remains unmodified for the lifetime of the stream object.

We recommend supplying a communicator instance, otherwise you will not be able to decode proxy objects. The stream also inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` methods:

Java

```
package com.zeroc.Ice;

public class InputStream
{
   public void initialize(Communicator communicator);
   public void initialize(Communicator communicator, EncodingVersion
encoding);
   ...
}
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following methods to manually configure the stream:

Java

```
package com.zeroc.Ice;

public class InputStream
{
    public void setValueFactoryManager(ValueFactoryManager vfm);
    public void setLogger(Logger logger);
    public void
setCompactIdResolver(java.util.function.IntFunction<String> r);
    public void setClassResolver(java.util.function.Function<String,
Class<?>> r);
    public void setSliceValues(bool);
    public void setTraceSlicing(bool);

    ...
}
```

The settings include:

- Value factory manager
  A value factory manager supplies custom factories for Slice class types.
- Logger
  The stream uses a logger to record warning and trace messages.
- Compact ID resolver
  A compact ID resolver for translating numeric values into Slice type IDs. The stream invokes the resolver by passing it the numeric compact ID. The resolver is expected to return the Slice type ID associated with that numeric ID or an empty string if the numeric ID is unknown. The application must supply an object that implements `IntFunction<String>`.
- Class resolver
  Translates Slice type IDs into Java classes. The resolver is expected to return the class corresponding to the Slice type ID or null if the ID is unknown. The application must supply an object that implements `Function<String, Class<?>>`. If you initialized the stream with a communicator, the stream uses the communicator's class resolver by default. The built-in class resolver supports package metadata and configuration properties that allow you to customize the Java mapping.
- Slice values
  The flag indicates whether to slice instances of Slice classes to a known Slice type when a more derived type is unknown. An

instance is "sliced" when no static information is available for a Slice type ID and no factory can be found for that type, resulting in the creation of an instance of a less-derived type. If slicing is disabled in this situation, the stream raises the exception `NoValueFactoryException`. The default behavior is to allow slicing.

- Trace slicing
  The flag indicates whether to log messages when instances of Slice classes are sliced. If the stream is initialized with a communicator, this setting defaults to the value of the `Ice.Trace.Slicing` property, otherwise the setting defaults to false.

Extracting from an `InputStream` in Java

`InputStream` provides a number of `read` methods that allow you to extract Slice types from the stream.

For example, you can extract a boolean and a sequence of strings from a stream as follows:

Java

```
import com.zeroc.Ice.InputStream;
...
byte[] data = ...
InputStream in = new InputStream(communicator, data);
boolean b = in.readBool();
String[] seq = in.readStringSeq();
```

Here are the methods for extracting data from a stream:

Java

```
package com.zeroc.Ice;

public class InputStream
{
    public boolean readBool();
    public java.util.Optional<Boolean> readBool(int tag);
    public boolean[] readBoolSeq();
    public java.util.Optional<boolean[]> readBoolSeq(int tag);

    public byte readByte();
    public java.util.Optional<Byte> readByte(int tag);
    public byte[] readByteSeq();
    public java.util.Optional<byte[]> readByteSeq(int tag);
    public java.nio.ByteBuffer readByteBuffer();

    public short readShort();
    public java.util.Optional<Short> readShort(int tag);
    public short[] readShortSeq();
    public java.util.Optional<short[]> readShortSeq(int tag);
    public java.nio.ShortBuffer readShortBuffer();

    public int readInt();
    public java.util.OptionalInt readInt(int tag);
    public int[] readIntSeq();
    public java.util.Optional<int[]> readIntSeq(int tag);
    public java.nio.IntBuffer readIntBuffer();
```

```
    public long readLong();
    public java.util.OptionalLong readLong(int tag);
    public long[] readLongSeq();
    public java.util.Optional<long[]> readLongSeq(int tag);
    public java.nio.LongBuffer readLongBuffer();

    public float readFloat();
    public java.util.Optional<Float> readFloat(int tag);
    public float[] readFloatSeq();
    public java.util.Optional<float[]> readFloatSeq(int tag);
    public java.nio.FloatBuffer readFloatBuffer();

    public double readDouble();
    public java.util.OptionalDouble readDouble(int tag);
    public double[] readDoubleSeq();
    public java.util.Optional<double[]> readDoubleSeq(int tag);
    public java.nio.DoubleBuffer readDoubleBuffer();

    public String readString();
    public java.util.Optional<String> readString(int tag);
    public String[] readStringSeq();
    public java.util.Optional<String[]> readStringSeq(int tag);

    public int readSize();
    public int readAndCheckSeqSize(int minSizeWireSize);

    public ObjectPrx readProxy();
    public java.util.Optional<ObjectPrx> readProxy(int tag);

    public void readValue(java.util.function.Consumer<Value> cb);
    public void readValue(int tag,
java.util.function.Consumer<java.util.Optional<Value>> cb);

    public int readEnum(int maxValue);

    public void throwException() throws UserException;
    public void throwException(UserExceptionFactory factory) throws
UserException;

    public void startValue();
    public SlicedData endValue(boolean preserve);

    public void startException();
    public SlicedData endException(boolean preserve);

    public String startSlice();
    public void endSlice();
    public void skipSlice();
```

```
public EncodingVersion startEncapsulation();
public void endEncapsulation();
public EncodingVersion skipEncapsulation();
public EncodingVersion skipEmptyEncapsulation();
public int getEncapsulationSize();

public EncodingVersion getEncoding();

public void readPendingValues();

public java.io.Serializable readSerializable();
public byte[] readBlob(int sz);

public void skip(int sz);
public void skipSize();

public boolean readOptional(int tag, OptionalFormat format);

public int pos();
```

```
    public void pos(int n);
}
```

Member functions are provided for extracting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `java.util.Optional<`*`Type`*`> read`*`Type`*`(int tag)`
  `java.util.Optional<`*`type`*`[]> read`*`Type`*`Seq(int tag)`
  Extracts an optional value, or an optional sequence, respectively. Call the `isPresent` method of the returned optional value to determine whether a value is set.

- `java.nio.`*`Type`*`Buffer read`*`Type`*`Buffer()`
  A more efficient way of extracting a sequence without making a copy of the data. The returned buffer object is read-only.

- `int readSize()`
  The Ice encoding has a compact representation to indicate size. This function extracts a size and returns it as an integer.

- `int readAndCheckSeqSize(int minWireSize)`
  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element. If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `ObjectPrx readProxy()`
  This function returns an instance of the base proxy type, `ObjectPrx`. The Slice compiler optionally generates helper functions to extract proxies of user-defined types.

- `void readValue(java.util.function.Consumer<Value> cb)`
  The Ice encoding for class instances requires extraction to occur in stages. The `readValue` function accepts a consumer object. When the class instance is available, the stream invokes `accept` on the consumer. The application must call `readPendingValues` on the stream to ensure that all instances are properly extracted. Note that applications rarely need to invoke this member function directly; the helper functions generated by the Slice compiler are easier to use.

- `int readEnum(int maxValue)`
  Unmarshals the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.

- `void throwException() throws UserException`
  `void throwException(UserExceptionFactory factory) throws UserException`
  These functions extract a user exception from the stream and throw it. If the stored exception is of an unknown type, the functions attempt to extract and throw a less-derived exception. If that also fails, an exception is thrown: for the 1.0 encoding, the exception is `UnmarshalOutOfBoundsException`, for the 1.1 encoding, the exception is `UnknownUserException`. You can optionally supply an object implementing `UserExceptionFactory`:

Java

```
package com.zeroc.Ice;

public interface UserExceptionFactory
{
    void createAndThrow(String typeId)
        throws UserException;
}
```

As the stream decodes each slice of a user exception, it will invoke the factory first to give it an opportunity to raise an exception corresponding to the given Slice type ID. If the factory does not recognize the type ID, it must return without throwing. In this case, the stream will make its own attempt to locate a class for the type ID; if that fails, the stream skips the slice and tries again for the next slice.

- `String startSlice()`
  `void endSlice()`
  `void skipSlice()`
  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of an class instance or user exception. The `startSlice` method returns the type ID of the next slice, which may be an empty string depending on the format used to encode the instance or exception.

- `void startValue()`
  `SlicedData endValue(boolean preserve)`
  The `startValue` method must be called prior to reading the slices of a class instance. The `endValue` method must be called after all slices have been read. Pass true to `endValue` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endValue` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the instance with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `void startException()`
  `SlicedData endException(boolean preserve)`
  The `startException` method must be called prior to reading the slices of an exception. The `endException` method must be called after all slices have been read. Pass true to `endException` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endException` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the exception with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `EncodingVersion startEncapsulation()`
  `void endEncapsulation()`
  `EncodingVersion skipEncapsulation()`
  `EncodingVersion skipEmptyEncapsulation()`
  Start, end, and skip an encapsulation, respectively. For methods that return `EncodingVersion`, the return value represents the encoding version used to encode the contents of the encapsulation.

- `int getEncapsulationSize()`
  Returns the size of the current encapsulation.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently in use by the stream.

- `void readPendingValues()`
  An application must call this function after all other data has been extracted, but only if class instances were encoded. This function extracts the state of class instances and invokes their corresponding callback objects (see `readValue`). For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `java.io.Serializable readSerializable()`
  Reads a serializable Java object from the stream.

- `byte[] readBlob(int sz)`
  Extracts an array of `sz` bytes from the stream at its current position. The returned byte array contains undecoded data from the stream's internal buffer. The stream's position is advanced by `sz`.

- `void skip(int sz)`
  Skips the given number of bytes.
- `void skipSize()`
  Reads a size at the current position and skips that number of bytes.
- `boolean readOptional(int tag, OptionalFormat fmt)`
  Returns true if an optional value with the given tag and format is present, or false otherwise. If this method returns true, the data associated with that optional value must be read next. Optional values must be read in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  Java

  ```
  package com.zeroc.Ice;

  public enum OptionalFormat
  {
      OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
  OptionalFormatF8,
      OptionalFormatSize, OptionalFormatVSize, OptionalFormatFSize,
      OptionalFormatEndMarker
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.
- `int pos()`
  `void pos(int n)`
  Returns or modifies the stream's current position, respectively.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in Java
- Stream Helper Methods in Java

The OutputStream Interface in Java

On this page:

Initializing an `OutputStream` in Java

The `OutputStream` class provides a number of overloaded constructors:

Java

```
package com.zeroc.Ice;

public class OutputStream
{
    public OutputStream();
    public OutputStream(boolean direct);
    public OutputStream(Communicator communicator);
    public OutputStream(Communicator communicator, boolean direct);
    public OutputStream(Communicator communicator, EncodingVersion
version);
    public OutputStream(Communicator communicator, EncodingVersion
version, boolean direct);

    ...
}
```

The constructors optionally accept the following arguments:

- A communicator instance
- An encoding version
- A flag indicating whether the stream's internal buffer should be allocated as a "direct" buffer.

We recommend supplying a communicator instance. The stream inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` methods:

Java

```
package com.zeroc.Ice;

public class OutputStream
{
    public void initialize(Communicator communicator);
    public void initialize(Communicator communicator, EncodingVersion
version);

    ...
}
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following method to manually configure the stream:

Java

```
package com.zeroc.Ice;

public class OutputStream
{
    public void setFormat(FormatType fmt);

    ...
}
```

For instances of Slice classes, the format determines how the slices of an instance are encoded. If the stream is initialized with a communicator, this setting defaults to the value of `com.zeroc.Ice.Default.SlicedFormat`, otherwise the setting defaults to the compact format.

Inserting into an `OutputStream` in Java

`OutputStream` provides a number of `write` methods that allow you to insert Slice types into the stream.

For example, you can insert a boolean and a sequence of strings into a stream as follows:

Java

```
import com.zeroc.Ice.OutputStream;
...
final String[] seq = { "Ice", "rocks!" };
OutputStream out = new OutputStream(communicator);
out.writeBool(true);
out.writeStringSeq(seq);
byte[] data = out.finished();
```

Here are the methods for inserting data into a stream:

Java

```
package com.zeroc.Ice;

public class OutputStream
{
    public void writeBool(boolean v);
    public void writeBool(int tag, boolean v);
    public void writeBool(int tag, java.util.Optional<Boolean> v);
    public void writeBoolSeq(boolean[] v);
    public void writeBoolSeq(int tag, boolean[] v);
    public void writeBoolSeq(int tag, java.util.Optional<boolean[]> v);
    public void rewriteBool(boolean v, int dest);

    public void writeByte(byte v);
    public void writeByte(int tag, byte v);
    public void writeByte(int tag, java.util.Optional<Byte> v);
    public void writeByteSeq(byte[] v);
    public void writeByteSeq(int tag, byte[] v);
    public void writeByteSeq(int tag, java.util.Optional<byte[]> v);
    public void rewriteByte(byte v, int dest);
    public void writeByteBuffer(java.nio.ByteBuffer v);

    public void writeShort(short v);
    public void writeShort(int tag, short v);
    public void writeShort(int tag, java.util.Optional<Short> v);
    public void writeShortSeq(short[] v);
    public void writeShortSeq(int tag, short[] v);
    public void writeShortSeq(int tag, java.util.Optional<short[]> v);
    public void writeShortBuffer(java.nio.ShortBuffer v);

    public void writeInt(int v);
    public void writeInt(int tag, int v);
    public void writeInt(int tag, java.util.OptionalInt v);
    public void writeIntSeq(int[] v);
    public void writeIntSeq(int tag, int[] v);
    public void writeIntSeq(int tag, java.util.Optional<int[]> v);
    public void rewriteInt(int v, int dest);
    public void writeIntBuffer(java.nio.IntBuffer v);

    public void writeLong(long v);
    public void writeLong(int tag, long v);
    public void writeLong(int tag, java.util.OptionalLong v);
    public void writeLongSeq(long[] v);
    public void writeLongSeq(int tag, long[] v);
    public void writeLongSeq(int tag, java.util.Optional<long[]> v);
    public void writeLongBuffer(java.nio.LongBuffer v);

    public void writeFloat(float v);
    public void writeFloat(int tag, float v);
    public void writeFloat(int tag, java.util.Optional<Float> v);
```

```
    public void writeFloatSeq(float[] v);
    public void writeFloatSeq(int tag, float[] v);
    public void writeFloatSeq(int tag, java.util.Optional<float[]> v);
    public void writeFloatBuffer(java.nio.FloatBuffer v);

    public void writeDouble(double v);
    public void writeDouble(int tag, double v);
    public void writeDouble(int tag, java.util.OptionalDouble v);
    public void writeDoubleSeq(double[] v);
    public void writeDoubleSeq(int tag, double[] v);
    public void writeDoubleSeq(int tag, java.util.Optional<double[]> v);
    public void writeDoubleBuffer(java.nio.DoubleBuffer v);

    public void writeString(String v);
    public void writeString(int tag, String v);
    public void writeString(int tag, java.util.Optional<String> v);
    public void writeStringSeq(String[] v);
    public void writeStringSeq(int tag, String[] v);
    public void writeStringSeq(int tag, java.util.Optional<String[]> v);

    public void writeSize(int sz);

    public void writeProxy(ObjectPrx v);
    public void writeProxy(int tag, ObjectPrx v);
    public void writeProxy(int tag, java.util.Optional<ObjectPrx> v);

    public void writeValue(Value v);
    public void writeValue(int tag, Value v);
    public <T extends Value> void writeValue(int tag,
java.util.Optional<T> v);

    public void writeEnum(int v, int maxValue);

    public void writeBlob(byte[] v);
    public void writeBlob(byte[] v, int off, int len);

    public void writeException(UserException ex);

    public void startValue(SlicedData sd);
    public void endValue();

    public void startException(SlicedData sd);
    public void endException();

    public void startSlice(String typeId, int compactId, boolean last);
    public void endSlice();

    public void startEncapsulation(EncodingVersion encoding, FormatType
format);
    public void startEncapsulation();
```

```
public void endEncapsulation();
public void writeEmptyEncapsulation(EncodingVersion encoding);
public void writeEncapsulation(byte[] v);

public EncodingVersion getEncoding();

public void writePendingValues();

public boolean writeOptional(int tag, OptionalFormat format);

public int pos();
public void pos(int n);

public int startSize();
public void endSize(int pos);

public byte[] finished();

public void writeSerializable(java.io.Serializable o);
```

```
    public void resize(int sz);
}
```

Member functions are provided for inserting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void write`*`Type`*`(int tag, `*`type`*` v)`
  `void write`*`Type`*`(int tag, java.util.Optional<`*`Type`*`> v)`
  `void write`*`Type`*`Seq(int tag, `*`type`*`[] v)`
  `void write`*`Type`*`Seq(int tag, java.util.Optional<`*`type`*`[]> v)`
  Inserts an optional value. Methods that accept `Optional` instances only insert a value if the argument is non-null and contains a value.

- `void write`*`Type`*`Buffer(java.nio.`*`Type`*`Buffer v)`
  Inserts the remaining contents of the given buffer as a sequence. A null value implies an empty sequence.

- `void rewriteByte(byte v, int dest)`
  `void rewriteBool(boolean v, int dest)`
  `void rewriteInt(int v, int dest)`
  Overwrites the byte(s) at an existing location in the buffer at the given destination with a value. These methods do not change the current position of the stream.

- `void writeSize(int sz)`
  The Ice encoding has a compact representation to indicate size. This function converts the given non-negative integer into the proper encoded representation.

- `void writeProxy(ObjectPrx v)`
  Inserts a proxy.

- `void writeValue(Value v)`
  Inserts an instance of a Slice class. The Ice encoding for class instances may cause the insertion to be delayed, in which case the stream retains a reference to the given instance and does not insert its state it until `writePendingValues` is invoked on the stream.

- `void writeEnum(int val, int maxValue)`
  Writes the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.

- `void writeBlob(byte[] v)`
  `void writeBlob(byte[] v, int off, int len)`
  Copies the given blob of bytes directly to the stream's internal buffer without modification.

- `void writeException(UserException ex)`
  Inserts a user exception.

- `void startValue(SlicedData sd)`
  `void endValue()`
  When marshaling the slices of a class instance, the application must first call `startValue`, then marshal the slices, and finally call `endValue`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or null if there are no preserved slices.

- `void startException(SlicedData sd)`
  `void endException()`
  When marshaling the slices of an exception, the application must first call `startException`, then marshal the slices, and finally call `endException`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or 0 if there are no preserved slices.

- `void startSlice(String typeId, int compactId, boolean last)`
  `void endSlice()`
  Starts and ends a slice of class or exception member data. The call to `startSlice` must include the type ID for the current slice, the corresponding compact ID for the type (if any), and a boolean indicating whether this is the last slice of the class instance or exception. The compact ID is only relevant for class instances; pass a negative value to indicate the encoding should use the string type ID.

- `void startEncapsulation(EncodingVersion encoding, FormatType format)`
  `void startEncapsulation()`
  `void endEncapsulation()`
  Starts and ends an encapsulation, respectively. The first overloading of `startEncapsulation` allows you to specify the encoding version as well as the format to use for any class instances and exceptions marshaled within this encapsulation.

- `void writeEmptyEncapsulation(EncodingVersion encoding)`
  Writes an encapsulation having the given encoding version with no encoded data.

- `void writeEncapsulation(byte[] v)`
  Copies the bytes representing an encapsulation from the given array into the stream.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently being used by the stream.

- `void writePendingValues()`
  Encodes the state of class instances whose insertion was delayed during `writeValue`. This member function must only be called once. For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `boolean writeOptional(int tag, OptionalFormat fmt)`
  Prepares the stream to write an optional value with the given tag and format. Returns true if the value should be written, or false otherwise. A return value of false indicates that the encoding version in use by the stream does not support optional values. If this method returns true, the data associated with that optional value must be written next. Optional values must be written in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  Java

  ```
  package com.zeroc.Ice;

  enum OptionalFormat
  {
      OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
  OptionalFormatF8,
      OptionalFormatSize, OptionalFormatVSize, OptionalFormatFSize,
      OptionalFormatEndMarker
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.

- `int pos()`
  `void pos(int n)`
  Returns or changes the stream's current position, respectively.

- `int startSize()`
  `void endSize(int n)`
  The encoding for optional values uses a 32-bit integer to hold the size of variable-length types. Calling `startSize` writes a placeholder value for the size and returns the starting position of the size value; after writing the data, call `endSize` to patch the placeholder with the actual size at the given position.

- `byte[] finished()`
  Indicates that marshaling is complete and returns the encoded byte sequence. This member function must only be called once.

- `void writeSerializable(java.io.Serializable v)`
  Writes a serializable Java object to the stream.

- `void resize(int sz)`
  Allocates space for `sz` more bytes in the buffer. The stream implementation internally uses this function prior to copying more data into the buffer.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in Java

Stream Helper Methods in Java

The stream classes provide all of the low-level methods necessary for encoding and decoding Ice types. However, it would be tedious and error-prone to manually encode complex Ice types such as classes, structs, and dictionaries using these low-level functions. For this reason, the Slice compiler generates helper methods for streaming complex Ice types.

We will use the following Slice definitions to demonstrate the language mapping:

Slice

```
module M
{
    sequence<...> Seq;
    dictionary<...> Dict;
    struct S
    {
        ...
    }
    enum E { ... }
    class C
    {
        ...
    }
    interface I
    {
        ...
    }
}
```

The Slice compiler generates the corresponding helper classes and methods shown below:

Java

```
package M;

public class SeqHelper
{
    public static T[] read(com.zeroc.Ice.InputStream in);
    public static void write(com.zeroc.Ice.OutputStream out, T[] v);

    public static java.util.Optional<T[]> read(com.zeroc.Ice.InputStream
in, int tag);
    public static void write(com.zeroc.Ice.OutputStream out, int tag,
java.util.Optional<T[]> v);
    public static void write(com.zeroc.Ice.OutputStream out, int tag,
T[] v);
    ...
}

public class DictHelper
{
```

```
    public static java.util.Map<..., ...> read(com.zeroc.Ice.InputStream
in);
    public static void write(com.zeroc.Ice.OutputStream out,
java.util.Map<..., ...> v);

    public static java.util.Optional<java.util.Map<..., ...>>
read(com.zeroc.Ice.InputStream in, int tag);
    public static void write(com.zeroc.Ice.OutputStream out, int tag,
java.util.Optional<java.util.Map<..., ...>> v);
    public static void write(com.zeroc.Ice.OutputStream out, int tag,
java.util.Map<..., ...> v);
    ...
}

public class S ...
{
    public static S ice_read(com.zeroc.Ice.InputStream in);
    public static void ice_write(com.zeroc.Ice.OutputStream out, S v);

    public static java.util.Optional<S>
ice_read(com.zeroc.Ice.InputStream in, int tag);
    public static void ice_write(com.zeroc.Ice.OutputStream out, int
tag, java.util.Optional<S> v);
    public static void ice_write(com.zeroc.Ice.OutputStream out, int
tag, S v);

    // Instance methods
    public void ice_readMembers(com.zeroc.Ice.InputStream in);
    public void ice_writeMembers(com.zeroc.Ice.OutputStream out);
    ...
}

public enum E ...
{
    public static E ice_read(com.zeroc.Ice.InputStream in);
    public static void ice_write(com.zeroc.Ice.OutputStream out, E v);

    public static java.util.Optional<E>
ice_read(com.zeroc.Ice.InputStream in, int tag);
    public static void ice_write(com.zeroc.Ice.OutputStream out, int
tag, java.util.Optional<E> v);
    public static void ice_write(com.zeroc.Ice.OutputStream out, int
tag, E v);

    // Instance method
```

```
    public void ice_write(com.zeroc.Ice.OutputStream out);
    ...
}
```

See Also

- Data Encoding
- slice2java Command-Line Options
- The InputStream Interface in Java
- Optional Values

Java Compat Streaming Interfaces

The stream API allows you to manually marshal and unmarshal Slice types using the Ice data encoding.

Topics

The InputStream Interface in Java Compat

On this page:

Initializing an `InputStream` in Java

The `InputStream` class provides a number of overloaded constructors:

Java

```
package Ice;

public class InputStream
{
    public InputStream();
    public InputStream(byte[] data);
    public InputStream(java.nio.ByteBuffer buf);
    public InputStream(Communicator communicator);
    public InputStream(Communicator communicator, byte[] data);
    public InputStream(Communicator communicator, java.nio.ByteBuffer
buf);
    public InputStream(EncodingVersion encoding);
    public InputStream(EncodingVersion encoding, byte[] data);
    public InputStream(EncodingVersion encoding, java.nio.ByteBuffer
buf);
    public InputStream(Communicator communicator, EncodingVersion
encoding);
    public InputStream(Communicator communicator, EncodingVersion
encoding, byte[] data);
    public InputStream(Communicator communicator, EncodingVersion
encoding, java.nio.ByteBuffer buf);

    ...
}
```

The constructors accept three types of arguments:

- A communicator instance
- An encoding version
- The encoded data that you intend to decode

You'll normally supply the encoded data argument, which the stream accepts as either an array of bytes or a `ByteBuffer`. In either case, the stream does not make a copy of the data; rather, it uses the data as supplied and assumes it remains unmodified for the lifetime of the stream object.

We recommend supplying a communicator instance, otherwise you will not be able to decode proxy objects. The stream also inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` methods:

Java

```
package Ice;

public class InputStream
{
   public void initialize(Communicator communicator);
   public void initialize(Communicator communicator, EncodingVersion
encoding);
   ...
}
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following methods to manually configure the stream:

Java

```
package Ice;

public class InputStream
{
    public void setValueFactoryManager(ValueFactoryManager vfm);
    public void setLogger(Logger logger);
    public void setCompactIdResolver(CompactIdResolver r);
    public void setClassResolver(ClassResolver r);
    public void setSliceValues(bool);
    public void setTraceSlicing(bool);

    ...
}
```

The settings include:

- Value factory manager
  A value factory manager supplies custom factories for Slice class types.
- Logger
  The stream uses a logger to record warning and trace messages.
- Compact ID resolver
  A compact ID resolver for translating numeric values into Slice type IDs. The stream invokes the resolver by passing it the numeric compact ID. The resolver is expected to return the Slice type ID associated with that numeric ID or an empty string if the numeric ID is unknown. The application must supply an object that implements `CompactIdResolver`:

Java

```
package Ice;

public interface CompactIdResolver
{
    String resolve(int id);
}
```

- Class resolver
  Translates Slice type IDs into Java classes. The resolver is expected to return the class corresponding to the Slice type ID or null if the ID is unknown. The application must supply an object that implements `ClassResolver`:

  Java

  ```
  package Ice;

  public interface ClassResolver
  {
      Class<?> resolveClass(String typeId);
  }
  ```

  If you initialized the stream with a communicator, the stream uses the communicator's class resolver by default. The built-in class resolver supports package metadata and configuration properties that allow you to customize the Java mapping.

- Slice values
  The flag indicates whether to slice instances of Slice classes to a known Slice type when a more derived type is unknown. An instance is "sliced" when no static information is available for a Slice type ID and no factory can be found for that type, resulting in the creation of an instance of a less-derived type. If slicing is disabled in this situation, the stream raises the exception `NoValueFactoryException`. The default behavior is to allow slicing.

- Trace slicing
  The flag indicates whether to log messages when instances of Slice classes are sliced. If the stream is initialized with a communicator, this setting defaults to the value of the `Ice.Trace.Slicing` property, otherwise the setting defaults to false.

Extracting from an `InputStream` in Java

`InputStream` provides a number of `read` methods that allow you to extract Slice types from the stream.

For example, you can extract a boolean and a sequence of strings from a stream as follows:

Java

```
byte[] data = ...
Ice.InputStream in = new Ice.InputStream(communicator, data);
boolean b = in.readBool();
String[] seq = in.readStringSeq();
```

Here are the methods for extracting data from a stream:

Java

```
package Ice;

public class InputStream
{
    public boolean readBool();
    public void readBool(int tag, BooleanOptional v);
    public boolean[] readBoolSeq();
    public void readBoolSeq(int tag, Optional<boolean[]> v);

    public byte readByte();
    public void readByte(int tag, ByteOptional v);
    public byte[] readByteSeq();
    public void readByteSeq(int tag, Optional<byte[]> v);
    public java.nio.ByteBuffer readByteBuffer();

    public short readShort();
    public void readShort(int tag, ShortOptional v);
    public short[] readShortSeq();
    public void readShortSeq(int tag, Optional<short[]> v);
    public java.nio.ShortBuffer readShortBuffer();

    public int readInt();
    public void readInt(int tag, IntOptional v);
    public int[] readIntSeq();
    public void readIntSeq(int tag, Optional<int[]> v);
    public java.nio.IntBuffer readIntBuffer();

    public long readLong();
    public void readLong(int tag, LongOptional v);
    public long[] readLongSeq();
    public void readLongSeq(int tag, Optional<long[]> v);
    public java.nio.LongBuffer readLongBuffer();

    public float readFloat();
    public void readFloat(int tag, FloatOptional v);
    public float[] readFloatSeq();
    public void readFloatSeq(int tag, Optional<float[]> v);
    public java.nio.FloatBuffer readFloatBuffer();

    public double readDouble();
    public void readDouble(int tag, DoubleOptional v);
    public double[] readDoubleSeq();
    public void readDoubleSeq(int tag, Optional<double[]> v);
    public java.nio.DoubleBuffer readDoubleBuffer();

    public String readString();
    public void readString(int tag, Optional<String> v);
    public String[] readStringSeq();
    public void readStringSeq(int tag, Optional<String[]> v);
```

```
    public int readSize();
    public int readAndCheckSeqSize(int minSizeWireSize);

    public ObjectPrx readProxy();
    public void readProxy(int tag, Optional<ObjectPrx> v);

    public void readValue(ReadValueCallback cb);
    public void readValue(int tag, Optional<Ice.Object> v);

    public int readEnum(int maxValue);

    public void throwException() throws UserException;
    public void throwException(UserExceptionFactory factory) throws
UserException;

    public void startValue();
    public SlicedData endValue(boolean preserve);

    public void startException();
    public SlicedData endException(boolean preserve);

    public String startSlice();
    public void endSlice();
    public void skipSlice();

    public EncodingVersion startEncapsulation();
    public void endEncapsulation();
    public EncodingVersion skipEncapsulation();
    public EncodingVersion skipEmptyEncapsulation();
    public int getEncapsulationSize();

    public EncodingVersion getEncoding();

    public void readPendingValues();

    public java.io.Serializable readSerializable();
    public byte[] readBlob(int sz);

    public void skip(int sz);
    public void skipSize();

    public boolean readOptional(int tag, OptionalFormat format);

    public int pos();
```

```
    public void pos(int n);
}
```

Member functions are provided for extracting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void read`*`Type`*`(int tag, `*`Type`*`Optional v)`
  `void read`*`Type`*`Seq(int tag, Optional<`*`type`*`[]> v)`
  Extracts an optional value, or an optional sequence, respectively. The given `Optional` parameter must be non-null; if the optional value is present in the stream, the parameter is marked as set and its value is also unmarshaled.
- `java.nio.`*`Type`*`Buffer read`*`Type`*`Buffer()`
  A more efficient way of extracting a sequence without making a copy of the data. The returned buffer object is read-only.
- `int readSize()`
  The Ice encoding has a compact representation to indicate size. This function extracts a size and returns it as an integer.
- `int readAndCheckSeqSize(int minWireSize)`
  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element. If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.
- `Ice.ObjectPrx readProxy()`
  This function returns an instance of the base proxy type, `ObjectPrx`. The Slice compiler optionally generates helper functions to extract proxies of user-defined types.
- `void readValue(ReadValueCallback cb)`
  The Ice encoding for class instances requires extraction to occur in stages. The `readValue` function accepts a callback object of type `ReadValueCallback`, whose definition is shown below:

  Java

  ```
  package Ice;

  public interface ReadValueCallback
  {
      void valueReady(Ice.Object obj);
  }
  ```

  When the class instance is available, the callback's valueReady member function is called. The application must call `readPendingValues` on the stream to ensure that all instances are properly extracted. Note that applications rarely need to invoke this member function directly; the helper functions generated by the Slice compiler are easier to use.
- `int readEnum(int maxValue)`
  Unmarshals the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.
- `void throwException() throws UserException`
  `void throwException(UserExceptionFactory factory) throws UserException`
  These functions extract a user exception from the stream and throw it. If the stored exception is of an unknown type, the functions

attempt to extract and throw a less-derived exception. If that also fails, an exception is thrown: for the 1.0 encoding, the exception is `UnmarshalOutOfBoundsException`, for the 1.1 encoding, the exception is `UnknownUserException`. You can optionally supply an object implementing `UserExceptionFactory`:

Java

```
package Ice;

public interface UserExceptionFactory
{
    void createAndThrow(String typeId)
        throws UserException;
}
```

As the stream decodes each slice of a user exception, it will invoke the factory first to give it an opportunity to raise an exception corresponding to the given Slice type ID. If the factory does not recognize the type ID, it must return without throwing. In this case, the stream will make its own attempt to locate a class for the type ID; if that fails, the stream skips the slice and tries again for the next slice.

- `String startSlice()`
  `void endSlice()`
  `void skipSlice()`
  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of an class instance or user exception. The `startSlice` method returns the type ID of the next slice, which may be an empty string depending on the format used to encode the instance or exception.

- `void startValue()`
  `SlicedData endValue(boolean preserve)`
  The `startValue` method must be called prior to reading the slices of a class instance. The `endValue` method must be called after all slices have been read. Pass true to `endValue` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endValue` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the instance with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `void startException()`
  `SlicedData endException(boolean preserve)`
  The `startException` method must be called prior to reading the slices of an exception. The `endException` method must be called after all slices have been read. Pass true to `endException` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endException` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the exception with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `EncodingVersion startEncapsulation()`
  `void endEncapsulation()`
  `EncodingVersion skipEncapsulation()`
  `EncodingVersion skipEmptyEncapsulation()`
  Start, end, and skip an encapsulation, respectively. For methods that return `EncodingVersion`, the return value represents the encoding version used to encode the contents of the encapsulation.

- `int getEncapsulationSize()`
  Returns the size of the current encapsulation.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently in use by the stream.

- `void readPendingValues()`
  An application must call this function after all other data has been extracted, but only if class instances were encoded. This function extracts the state of class instances and invokes their corresponding callback objects (see `readValue`). For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `java.io.Serializable readSerializable()`
  Reads a serializable Java object from the stream.

- `byte[] readBlob(int sz)`
  Extracts an array of `sz` bytes from the stream at its current position. The returned byte array contains undecoded data from the stream's internal buffer. The stream's position is advanced by `sz`.
- `void skip(int sz)`
  Skips the given number of bytes.
- `void skipSize()`
  Reads a size at the current position and skips that number of bytes.
- `boolean readOptional(int tag, OptionalFormat fmt)`
  Returns true if an optional value with the given tag and format is present, or false otherwise. If this method returns true, the data associated with that optional value must be read next. Optional values must be read in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  Java

  ```
  package Ice;
  public enum OptionalFormat
  {
      OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
  OptionalFormatF8,
      OptionalFormatSize, OptionalFormatVSize, OptionalFormatFSize,
      OptionalFormatEndMarker
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.
- `int pos()`
  `void pos(int n)`
  Returns or modifies the stream's current position, respectively.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in Java
- Stream Helper Methods in Java Compat

The OutputStream Interface in Java Compat

On this page:

Initializing an `OutputStream` in Java

The `OutputStream` class provides a number of overloaded constructors:

Java

```
package Ice;

public class OutputStream
{
    public OutputStream();
    public OutputStream(boolean direct);
    public OutputStream(Communicator communicator);
    public OutputStream(Communicator communicator, boolean direct);
    public OutputStream(Communicator communicator, EncodingVersion
version);
    public OutputStream(Communicator communicator, EncodingVersion
version, boolean direct);

    ...
}
```

The constructors optionally accept the following arguments:

- A communicator instance
- An encoding version
- A flag indicating whether the stream's internal buffer should be allocated as a "direct" buffer.

We recommend supplying a communicator instance. The stream inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` methods:

Java

```
package Ice;

public class OutputStream
{
    public void initialize(Communicator communicator);
    public void initialize(Communicator communicator, EncodingVersion
version);

    ...
}
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following method to manually configure the stream:

Java

```
package Ice;

public class OutputStream
{
    public void setFormat(FormatType fmt);

    ...
}
```

For instances of Slice classes, the format determines how the slices of an instance are encoded. If the stream is initialized with a communicator, this setting defaults to the value of `Ice.Default.SlicedFormat`, otherwise the setting defaults to the compact format.

Inserting into an `OutputStream` in Java

`OutputStream` provides a number of `write` methods that allow you to insert Slice types into the stream.

For example, you can insert a boolean and a sequence of strings into a stream as follows:

Java

```
final String[] seq = { "Ice", "rocks!" };
Ice.OutputStream out = new Ice.OutputStream(communicator);
out.writeBool(true);
out.writeStringSeq(seq);
byte[] data = out.finished();
```

Here are the methods for inserting data into a stream:

Java

```
package Ice;
```

```
public class OutputStream
{
    public void writeBool(boolean v);
    public void writeBool(int tag, boolean v);
    public void writeBool(int tag, BooleanOptional v);
    public void writeBoolSeq(boolean[] v);
    public void writeBoolSeq(int tag, boolean[] v);
    public void writeBoolSeq(int tag, Optional<boolean[]> v);
    public void rewriteBool(boolean v, int dest);

    public void writeByte(byte v);
    public void writeByte(int tag, byte v);
    public void writeByte(int tag, ByteOptional v);
    public void writeByteSeq(byte[] v);
    public void writeByteSeq(int tag, byte[] v);
    public void writeByteSeq(int tag, Optional<byte[]> v);
    public void rewriteByte(byte v, int dest);
    public void writeByteBuffer(java.nio.ByteBuffer v);

    public void writeShort(short v);
    public void writeShort(int tag, short v);
    public void writeShort(int tag, ShortOptional v);
    public void writeShortSeq(short[] v);
    public void writeShortSeq(int tag, short[] v);
    public void writeShortSeq(int tag, Optional<short[]> v);
    public void writeShortBuffer(java.nio.ShortBuffer v);

    public void writeInt(int v);
    public void writeInt(int tag, int v);
    public void writeInt(int tag, IntOptional v);
    public void writeIntSeq(int[] v);
    public void writeIntSeq(int tag, int[] v);
    public void writeIntSeq(int tag, Optional<int[]> v);
    public void rewriteInt(int v, int dest);
    public void writeIntBuffer(java.nio.IntBuffer v);

    public void writeLong(long v);
    public void writeLong(int tag, long v);
    public void writeLong(int tag, LongOptional v);
    public void writeLongSeq(long[] v);
    public void writeLongSeq(int tag, long[] v);
    public void writeLongSeq(int tag, Optional<long[]> v);
    public void writeLongBuffer(java.nio.LongBuffer v);

    public void writeFloat(float v);
    public void writeFloat(int tag, float v);
    public void writeFloat(int tag, FloatOptional v);
    public void writeFloatSeq(float[] v);
    public void writeFloatSeq(int tag, float[] v);
```

```
    public void writeFloatSeq(int tag, Optional<float[]> v);
    public void writeFloatBuffer(java.nio.FloatBuffer v);

    public void writeDouble(double v);
    public void writeDouble(int tag, double v);
    public void writeDouble(int tag, DoubleOptional v);
    public void writeDoubleSeq(double[] v);
    public void writeDoubleSeq(int tag, double[] v);
    public void writeDoubleSeq(int tag, Optional<double[]> v);
    public void writeDoubleBuffer(java.nio.DoubleBuffer v);

    public void writeString(String v);
    public void writeString(int tag, String v);
    public void writeString(int tag, Optional<String> v);
    public void writeStringSeq(String[] v);
    public void writeStringSeq(int tag, String[] v);
    public void writeStringSeq(int tag, Optional<String[]> v);

    public void writeSize(int sz);

    public void writeProxy(ObjectPrx v);
    public void writeProxy(int tag, ObjectPrx v);
    public void writeProxy(int tag, Optional<ObjectPrx> v);

    public void writeValue(Ice.Object v);
    public void writeValue(int tag, Ice.Object v);
    public <T extends Ice.Object> void writeValue(int tag, Optional<T>
v);

    public void writeEnum(int v, int maxValue);

    public void writeBlob(byte[] v);
    public void writeBlob(byte[] v, int off, int len);

    public void writeException(UserException ex);

    public void startValue(SlicedData sd);
    public void endValue();

    public void startException(SlicedData sd);
    public void endException();

    public void startSlice(String typeId, int compactId, boolean last);
    public void endSlice();

    public void startEncapsulation(EncodingVersion encoding, FormatType
format);
    public void startEncapsulation();
    public void endEncapsulation();
    public void writeEmptyEncapsulation(EncodingVersion encoding);
```

```
    public void writeEncapsulation(byte[] v);

    public EncodingVersion getEncoding();

    public void writePendingValues();

    public boolean writeOptional(int tag, OptionalFormat format);

    public int pos();
    public void pos(int n);

    public int startSize();
    public void endSize(int pos);

    public byte[] finished();

    public void writeSerializable(java.io.Serializable o);

    public void resize(int sz);
}
```

Member functions are provided for inserting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void write`*`Type`*`(int tag, `*`type`*` v)`
  `void write`*`Type`*`(int tag, `*`Type`*`Optional v)`
  `void write`*`Type`*`Seq(int tag, `*`type`*`[] v)`
  `void write`*`Type`*`Seq(int tag, Optional<`*`type`*`[]> v)`
  Inserts an optional value. Methods that accept `Optional` instances only insert a value if the argument is non-null and contains a value.

- `void write`*`Type`*`Buffer(java.nio.`*`Type`*`Buffer v)`
  Inserts the remaining contents of the given buffer as a sequence. A null value implies an empty sequence.

- `void rewriteByte(byte v, int dest)`
  `void rewriteBool(boolean v, int dest)`
  `void rewriteInt(int v, int dest)`
  Overwrites the byte(s) at an existing location in the buffer at the given destination with a value. These methods do not change the current position of the stream.

- `void writeSize(int sz)`
  The Ice encoding has a compact representation to indicate size. This function converts the given non-negative integer into the proper encoded representation.

- `void writeProxy(Ice.ObjectPrx v)`
  Inserts a proxy.

- `void writeValue(Ice.Object v)`
  Inserts an instance of a Slice class. The Ice encoding for class instances may cause the insertion to be delayed, in which case the stream retains a reference to the given instance and does not insert its state it until `writePendingValues` is invoked on the stream.

- `void writeEnum(int val, int maxValue)`
  Writes the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

Slice

```
enum Color { red, green, blue }
enum Fruit { Apple, Pear=3, Orange }
```

The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.

- `void writeBlob(byte[] v)`
  `void writeBlob(byte[] v, int off, int len)`
  Copies the given blob of bytes directly to the stream's internal buffer without modification.
- `void writeException(UserException ex)`
  Inserts a user exception.
- `void startValue(SlicedData sd)`
  `void endValue()`
  When marshaling the slices of a class instance, the application must first call `startValue`, then marshal the slices, and finally call `endValue`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or null if there are no preserved slices.
- `void startException(SlicedData sd)`
  `void endException()`
  When marshaling the slices of an exception, the application must first call `startException`, then marshal the slices, and finally call `endException`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or 0 if there are no preserved slices.
- `void startSlice(String typeId, int compactId, boolean last)`
  `void endSlice()`
  Starts and ends a slice of class or exception member data. The call to `startSlice` must include the type ID for the current slice, the corresponding compact ID for the type (if any), and a boolean indicating whether this is the last slice of the class instance or exception. The compact ID is only relevant for class instances; pass a negative value to indicate the encoding should use the string type ID.
- `void startEncapsulation(EncodingVersion encoding, FormatType format)`
  `void startEncapsulation()`
  `void endEncapsulation()`
  Starts and ends an encapsulation, respectively. The first overloading of `startEncapsulation` allows you to specify the encoding version as well as the format to use for any class instances and exceptions marshaled within this encapsulation.
- `void writeEmptyEncapsulation(EncodingVersion encoding)`
  Writes an encapsulation having the given encoding version with no encoded data.
- `void writeEncapsulation(byte[] v)`
  Copies the bytes representing an encapsulation from the given array into the stream.
- `EncodingVersion getEncoding()`
  Returns the encoding version currently being used by the stream.
- `void writePendingValues()`
  Encodes the state of class instances whose insertion was delayed during `writeValue`. This member function must only be called once. For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.
- `boolean writeOptional(int tag, OptionalFormat fmt)`
  Prepares the stream to write an optional value with the given tag and format. Returns true if the value should be written, or false otherwise. A return value of false indicates that the encoding version in use by the stream does not support optional values. If this method returns true, the data associated with that optional value must be written next. Optional values must be written in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

Java

```
package Ice;
enum OptionalFormat
{
    OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
OptionalFormatF8,
    OptionalFormatSize, OptionalFormatVSize, OptionalFormatFSize,
    OptionalFormatEndMarker
}
```

Refer to the encoding discussion for more information on the meaning of these values.

- `int pos()`
  `void pos(int n)`
  Returns or changes the stream's current position, respectively.
- `int startSize()`
  `void endSize(int n)`
  The encoding for optional values uses a 32-bit integer to hold the size of variable-length types. Calling `startSize` writes a placeholder value for the size and returns the starting position of the size value; after writing the data, call `endSize` to patch the placeholder with the actual size at the given position.
- `byte[] finished()`
  Indicates that marshaling is complete and returns the encoded byte sequence. This member function must only be called once.
- `void writeSerializable(java.io.Serializable v)`
  Writes a serializable Java object to the stream.
- `void resize(int sz)`
  Allocates space for `sz` more bytes in the buffer. The stream implementation internally uses this function prior to copying more data into the buffer.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in Java Compat

Stream Helper Methods in Java Compat

The stream classes provide all of the low-level methods necessary for encoding and decoding Ice types. However, it would be tedious and error-prone to manually encode complex Ice types such as classes, structs, and dictionaries using these low-level functions. For this reason, the Slice compiler generates helper methods for streaming complex Ice types.

We will use the following Slice definitions to demonstrate the language mapping:

Slice

```
module M
{
    sequence<...> Seq;
    dictionary<...> Dict;
    struct S
    {
        ...
    }
    enum E { ... }
    class C
    {
        ...
    }
    interface I
    {
        ...
    }
}
```

The Slice compiler generates the corresponding helper classes and methods shown below:

Java Compat

```
package M;

public class SeqHelper
{
    public static T[] read(Ice.InputStream in);
    public static void write(Ice.OutputStream out, T[] v);
    ...
}

public class DictHelper
{
    public static java.util.Map<..., ...> read(Ice.InputStream in);
    public static void write(Ice.OutputStream out, java.util.Map<...,
...> v);
    ...
}

public class S ...
```

```
{
    public static S ice_read(Ice.InputStream in);
    public static void ice_write(Ice.OutputStream out, S v);

    // Instance methods
    public void ice_readMembers(Ice.InputStream in);
    public void ice_writeMembers(Ice.OutputStream out);
    ...
}

public enum E ...
{
    public static E ice_read(Ice.InputStream in);
    public static void ice_write(Ice.OutputStream out, E v);

    // Instance method
    public void ice_write(Ice.OutputStream out);
    ...
}

public class CHolder implements Ice.ReadValueCallback ...
{
    public CHolder();
    public CHolder(C v);

    public C value;

    ...
}

public class IPrxHelper ...
{
    public static IPrx read(Ice.InputStream in);
```

```
    public static void write(Ice.OutputStream out, IPrx v);
    ...
}
```

See Also

- Data Encoding
- slice2java Command-Line Options
- The InputStream Interface in Java Compat
- Optional Values

C-Sharp Streaming Interfaces

The stream API allows you to manually marshal and unmarshal Slice types using the Ice data encoding.

Topics

The InputStream Interface in C-Sharp

On this page:

Initializing an `InputStream` in C#

The `InputStream` class provides a number of overloaded constructors:

C#

```
namespace Ice
{
    public class InputStream
    {

        public InputStream();
        public InputStream(byte[] data);
        public InputStream(Communicator communicator);
        public InputStream(Communicator communicator, byte[] data);
        public InputStream(EncodingVersion encoding);
        public InputStream(EncodingVersion encoding, byte[] data);
        public InputStream(Communicator communicator, EncodingVersion
encoding);
        public InputStream(Communicator communicator, EncodingVersion
encoding, byte[] data);
        ...
    }
}
```

The constructors accept three types of arguments:

- A communicator instance
- An encoding version
- A byte array containing the encoded data that you intend to decode

You'll normally supply the encoded data argument. The stream does not make a copy of this data; rather, it uses the data as supplied and assumes it remains unmodified for the lifetime of the stream object.

We recommend supplying a communicator instance, otherwise you will not be able to decode proxy objects. The stream also inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` methods:

C#

```
namespace Ice
{
    public class InputStream
    {
       public void initialize(Communicator communicator);
       public void initialize(Communicator communicator, EncodingVersion
encoding);
       ...
    }
}
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following methods to manually configure the stream:

C#

```
namespace Ice
{
    public class InputStream
    {
        public void setValueFactoryManager(ValueFactoryManager vfm);
        public void setLogger(Logger logger);
        public void setCompactIdResolver(System.Func<int, string> r);
        public void setClassResolver(System.Func<string, Type> r);
        public void setSliceValues(bool);
        public void setTraceSlicing(bool);

        ...
    }
}
```

The settings include:

- Value factory manager
  A value factory manager supplies custom factories for Slice class types.
- Logger
  The stream uses a logger to record warning and trace messages.
- Compact ID resolver
  A compact ID resolver for translating numeric values into Slice type IDs. The stream invokes the resolver by passing it the numeric compact ID. The resolver is expected to return the Slice type ID associated with that numeric ID or an empty string if the numeric ID is unknown. The application must supply a `System.Func<int, string>` delegate.
- Class resolver
  Translates Slice type IDs into C# classes. The resolver is expected to return the class corresponding to the Slice type ID or null if the ID is unknown. The application must supply a `System.Func<string, Type>` delegate. If you initialized the stream with a communicator, the stream uses the communicator's class resolver by default.
- Slice values
  The flag indicates whether to slice instances of Slice classes to a known Slice type when a more derived type is unknown. An instance is "sliced" when no static information is available for a Slice type ID and no factory can be found for that type, resulting in

the creation of an instance of a less-derived type. If slicing is disabled in this situation, the stream raises the exception `NoValueFactoryException`. The default behavior is to allow slicing.

- Trace slicing
  The flag indicates whether to log messages when instances of Slice classes are sliced. If the stream is initialized with a communicator, this setting defaults to the value of the `Ice.Trace.Slicing` property, otherwise the setting defaults to false.

Extracting from an `InputStream` in C#

`InputStream` provides a number of `read` methods that allow you to extract Slice types from the stream.

For example, you can extract a boolean and a sequence of strings from a stream as follows:

C#

```
byte[] data = ...
Ice.InputStream in = new Ice.InputStream(communicator, data);
bool b = in.readBool();
string[] seq = in.readStringSeq();
```

Here are the methods for extracting data from a stream:

C#

```
namespace Ice
{
    public class InputStream
    {

        public byte readByte();
        public Ice.Optional<byte> readByte(int tag);
        public void readByte(int tag, out bool isset, out byte v);
        public byte[] readByteSeq();
        public void readByteSeq(out List<byte> l);
        public void readByteSeq(out LinkedList<byte> l);
        public void readByteSeq(out Queue<byte> l);
        public void readByteSeq(out Stack<byte> l);
        public Ice.Optional<byte[]> readByteSeq(int tag);
        public void readByteSeq(int tag, out bool isset, out byte[] v);

        public bool readBool();
        public Ice.Optional<bool> readBool(int tag);
        public void readBool(int tag, out bool isset, out bool v);
        public bool[] readBoolSeq();
        public void readBoolSeq(out List<bool> l);
        public void readBoolSeq(out LinkedList<bool> l);
        public void readBoolSeq(out Queue<bool> l);
        public void readBoolSeq(out Stack<bool> l);
        public Ice.Optional<bool[]> readBoolSeq(int tag);
        public void readBoolSeq(int tag, out bool isset, out bool[] v);

        public short readShort();
        public Ice.Optional<short> readShort(int tag);
```

```
        public void readShort(int tag, out bool isset, out short v);
        public short[] readShortSeq();
        public void readShortSeq(out List<short> l);
        public void readShortSeq(out LinkedList<short> l);
        public void readShortSeq(out Queue<short> l);
        public void readShortSeq(out Stack<short> l);
        public Ice.Optional<short[]> readShortSeq(int tag);
        public void readShortSeq(int tag, out bool isset, out short[]
v);

        public int readInt();
        public Ice.Optional<int> readInt(int tag);
        public void readInt(int tag, out bool isset, out int v);
        public int[] readIntSeq();
        public void readIntSeq(out List<int> l);
        public void readIntSeq(out LinkedList<int> l);
        public void readIntSeq(out Queue<int> l);
        public void readIntSeq(out Stack<int> l);
        public Ice.Optional<int[]> readIntSeq(int tag);
        public void readIntSeq(int tag, out bool isset, out int[] v);

        public long readLong();
        public Ice.Optional<long> readLong(int tag);
        public void readLong(int tag, out bool isset, out long v);
        public long[] readLongSeq();
        public void readLongSeq(out List<long> l);
        public void readLongSeq(out LinkedList<long> l);
        public void readLongSeq(out Queue<long> l);
        public void readLongSeq(out Stack<long> l);
        public Ice.Optional<long[]> readLongSeq(int tag);
        public void readLongSeq(int tag, out bool isset, out long[] v);

        public float readFloat();
        public Ice.Optional<float> readFloat(int tag);
        public void readFloat(int tag, out bool isset, out float v);
        public float[] readFloatSeq();
        public void readFloatSeq(out List<float> l);
        public void readFloatSeq(out LinkedList<float> l);
        public void readFloatSeq(out Queue<float> l);
        public void readFloatSeq(out Stack<float> l);
        public Ice.Optional<float[]> readFloatSeq(int tag);
        public void readFloatSeq(int tag, out bool isset, out float[]
v);

        public double readDouble();
        public Ice.Optional<double> readDouble(int tag);
        public void readDouble(int tag, out bool isset, out double v);
        public double[] readDoubleSeq();
        public void readDoubleSeq(out List<double> l);
        public void readDoubleSeq(out LinkedList<double> l);
```

```
        public void readDoubleSeq(out Queue<double> l);
        public void readDoubleSeq(out Stack<double> l);
        public Ice.Optional<double[]> readDoubleSeq(int tag);
        public void readDoubleSeq(int tag, out bool isset, out double[]
v);

        public string readString();
        public Ice.Optional<string> readString(int tag);
        public void readString(int tag, out bool isset, out string v);
        public string[] readStringSeq();
        public void readStringSeq(out List<string> l);
        public void readStringSeq(out LinkedList<string> l);
        public void readStringSeq(out Queue<string> l);
        public void readStringSeq(out Stack<string> l);
        public Ice.Optional<string[]> readStringSeq(int tag);
        public void readStringSeq(int tag, out bool isset, out string[]
v);

        public int readSize();
        public int readAndCheckSeqSize(int minSizeWireSize);

        public ObjectPrx readProxy();
        public Ice.Optional<Ice.ObjectPrx> readProxy(int tag);
        public void readProxy(int tag, out bool isset, out Ice.ObjectPrx
v);

        public void readValue<T>(System.Action<T> cb) where T :
Ice.Value;
        public void readValue<T>(int tag, System.Action<T> cb) where T :
Ice.Value;
        public void readValue(System.Action<Ice.Value> cb);
        public void readValue(int tag, System.Action<Ice.Value> cb);

        public int readEnum(int maxValue);

        public void throwException();
        public void throwException(UserExceptionFactory factory);

        public void startValue();
        public SlicedData endValue(bool preserve);

        public void startException();
        public SlicedData endException(bool preserve);

        public string startSlice();
        public void endSlice();
        public void skipSlice();

        public EncodingVersion startEncapsulation();
        public void endEncapsulation();
```

```
public EncodingVersion skipEncapsulation();
public EncodingVersion skipEmptyEncapsulation();
public int getEncapsulationSize();

public EncodingVersion getEncoding();

public void readPendingValues();

public object readSerializable();
public void readBlob(byte[] v);
public byte[] readBlob(int sz);

public void skip(int sz);
public void skipSize();

public bool readOptional(int tag, OptionalFormat format);

public int pos();
public void pos(int n);
```

```
        ...
    }
}
```

Member functions are provided for extracting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `Ice.Optional<`*type*`> read`*Type*`(int tag)`
  `void read`*Type*`(int tag, out bool isset, out `*type*` v)`
  Extracts an optional value. The first method returns an `Optional` object that the caller can query to determine whether the value has been set and then obtain its value. The second method sets `isset` to true if the value is present and provides the value in `v`; if the value is not present, v is initialized to a default value appropriate for its type.

- `Ice.Optional<byte[]> readByteSeq(int tag)`
  `void readByteSeq(int tag, out bool isset, out byte[] v)`
  Extracts an optional sequence value. The first method returns an `Optional` object that the caller can query to determine whether the value has been set and then obtain its value. The second method sets `isset` to true if the value is present and provides the value in `v`; if the value is not present, v is initialized to null.

- `int readSize()`
  The Ice encoding has a compact representation to indicate size. This function extracts a size and returns it as an integer.

- `int readAndCheckSeqSize(int minWireSize)`
  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element. If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `Ice.ObjectPrx readProxy()`
  This function returns an instance of the base proxy type, `ObjectPrx`. The Slice compiler optionally generates helper functions to extract proxies of user-defined types.

- `void readValue<T>(System.Action<T> cb) where T : Ice.Value`
  The Ice encoding for class instances requires extraction to occur in stages. The `readValue` function accepts a delegate of type `System.Action<T>`. When the class instance is available, the delegate is called. The application must call `readPendingValues` on the stream to ensure that all instances are properly extracted.

- `int readEnum(int maxValue)`
  Unmarshals the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.

- `void throwException()`
  `void throwException(UserExceptionFactory factory)`
  These functions extract a user exception from the stream and throw it. If the stored exception is of an unknown type, the functions attempt to extract and throw a less-derived exception. If that also fails, an exception is thrown: for the 1.0 encoding, the exception is `UnmarshalOutOfBoundsException`, for the 1.1 encoding, the exception is `UnknownUserException`. You can optionally supply a `UserExceptionFactory` delegate:

C#

```
namespace Ice
{
    public delegate void UserExceptionFactory(string id);
}
```

As the stream decodes each slice of a user exception, it will invoke the factory first to give it an opportunity to raise an exception corresponding to the given Slice type ID. If the factory does not recognize the type ID, it must return without throwing. In this case, the stream will make its own attempt to locate a class for the type ID; if that fails, the stream skips the slice and tries again for the next slice.

- `string startSlice()`
  `void endSlice()`
  `void skipSlice()`
  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of a class instance or user exception. The `startSlice` method returns the type ID of the next slice, which may be an empty string depending on the format used to encode the instance or exception.

- `void startValue()`
  `SlicedData endValue(bool preserve)`
  The `startValue` method must be called prior to reading the slices of a class instance. The `endValue` method must be called after all slices have been read. Pass true to `endValue` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endValue` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the instance with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `void startException()`
  `SlicedData endException(bool preserve)`
  The `startException` method must be called prior to reading the slices of an exception. The `endException` method must be called after all slices have been read. Pass true to `endException` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endException` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the exception with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `EncodingVersion startEncapsulation()`
  `void endEncapsulation()`
  `EncodingVersion skipEncapsulation()`
  Start, end, and skip an encapsulation, respectively. The `startEncapsulation` and `skipEncapsulation` methods return the encoding version used to encode the contents of the encapsulation.

- `EncodingVersion skipEmptyEncapsulation()`
  Skips an encapsulation that is expected to be empty and returns its encoding version. The stream raises `EncapsulationException` if the encapsulation is not empty.

- `int getEncapsulationSize()`
  Returns the size of the current encapsulation.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently in use by the stream.

- `void readPendingValues()`
  An application must call this function after all other data has been extracted, but only if class instances were encoded. This function extracts the state of class instances and invokes their corresponding callback objects (see `readValue`). For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `object readSerializable()`
  Reads a serializable object from the stream.

- `byte[] readBlob(int sz)`
  Extracts an array of `sz` bytes from the stream at its current position. The returned byte array contains undecoded data from the stream's internal buffer. The stream's position is advanced by `sz`.

- `void readBlob(byte[] v)`
  Extracts an array of `v.Length` bytes from the stream at its current position. The byte array will contain undecoded data from the stream's internal buffer. The stream's position is advanced by `v.Length`.
- `void skip(int sz)`
  Skips the given number of bytes.
- `void skipSize()`
  Reads a size at the current position and skips that number of bytes.
- `bool readOptional(int tag, OptionalFormat fmt)`
  Returns true if an optional value with the given tag and format is present, or false otherwise. If this method returns true, the data associated with that optional value must be read next. Optional values must be read in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  C#

  ```
  namespace Ice
  {
      public enum OptionalFormat
      {
          OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
  OptionalFormatF8,
          OptionalFormatSize, OptionalFormatVSize,
  OptionalFormatFSize,
          OptionalFormatEndMarker
      }
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.
- `int pos()`
  `void pos(int n)`
  Returns or modifies the stream's current position, respectively.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in C-Sharp
- Stream Helper Methods in C-Sharp

The OutputStream Interface in C-Sharp

On this page:

Initializing an `OutputStream` in C#

The `OutputStream` class provides a number of overloaded constructors:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public OutputStream();
        public OutputStream(Communicator communicator);
        public OutputStream(Communicator communicator, EncodingVersion
version);

        ...
    }
}
```

The constructors optionally accept the following arguments:

- A communicator instance
- An encoding version

We recommend supplying a communicator instance. The stream inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` methods:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public void initialize(Communicator communicator);
        public void initialize(Communicator communicator,
EncodingVersion version);

        ...
    }
}
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following method to manually configure the stream:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public void setFormat(FormatType fmt);

        ...
    }
}
```

For instances of Slice classes, the format determines how the slices of an instance are encoded. If the stream is initialized with a communicator, this setting defaults to the value of `Ice.Default.SlicedFormat`, otherwise the setting defaults to the compact format.

Inserting into an `OutputStream` in C#

`OutputStream` provides a number of `write` methods that allow you to insert Slice types into the stream.

For example, you can insert a boolean and a sequence of strings into a stream as follows:

C#

```
string[] seq = { "Ice", "rocks!" };
Ice.OutputStream out = new Ice.OutputStream(communicator);
out.writeBool(true);
out.writeStringSeq(seq);
byte[] data = out.finished();
```

Here are the methods for inserting data into a stream:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public void writeBool(bool v);
        public void writeBool(int tag, bool v);
        public void writeBool(int tag, Optional<bool> v);
        public void writeBoolSeq(bool[] v);
        public void writeBoolSeq(int count, IEnumerable<bool> v);
        public void writeBoolSeq(int tag, bool[] v);
        public void writeBoolSeq(int tag, Optional<bool[]> v);
        public void writeBoolSeq<T>(int tag, int count, Ice.Optional<T>
v)
            where T : IEnumerable<bool>;
        public void writeBoolSeq(int tag, int count, IEnumerable<bool>
v);
```

```
        public void rewriteBool(bool v, int dest);

        public void writeByte(byte v);
        public void writeByte(int tag, byte v);
        public void writeByte(int tag, Optional<byte> v);
        public void writeByteSeq(byte[] v);
        public void writeByteSeq(int count, IEnumerable<byte> v);
        public void writeByteSeq(int tag, byte[] v);
        public void writeByteSeq(int tag, Optional<byte[]> v);
        public void writeByteSeq<T>(int tag, int count, Ice.Optional<T>
v)
            where T : IEnumerable<byte>;
        public void writeByteSeq(int tag, int count, IEnumerable<byte>
v);
        public void rewriteByte(byte v, int dest);

        public void writeShort(short v);
        public void writeShort(int tag, short v);
        public void writeShort(int tag, Optional<short> v);
        public void writeShortSeq(short[] v);
        public void writeShortSeq(int count, IEnumerable<short> v);
        public void writeShortSeq(int tag, short[] v);
        public void writeShortSeq(int tag, Optional<short[]> v);
        public void writeShortSeq<T>(int tag, int count, Ice.Optional<T>
v)
            where T : IEnumerable<short>;
        public void writeShortSeq(int tag, int count, IEnumerable<short>
v);

        public void writeInt(int v);
        public void writeInt(int tag, int v);
        public void writeInt(int tag, Optional<int> v);
        public void writeIntSeq(int[] v);
        public void writeIntSeq(int count, IEnumerable<int> v);
        public void writeIntSeq(int tag, int[] v);
        public void writeIntSeq(int tag, Optional<int[]> v);
        public void writeIntSeq<T>(int tag, int count, Ice.Optional<T>
v)
            where T : IEnumerable<int>;
        public void writeIntSeq(int tag, int count, IEnumerable<int> v);
        public void rewriteInt(int v, int dest);

        public void writeLong(long v);
        public void writeLong(int tag, long v);
        public void writeLong(int tag, Optional<long> v);
        public void writeLongSeq(long[] v);
        public void writeLongSeq(int count, IEnumerable<long> v);
        public void writeLongSeq(int tag, long[] v);
        public void writeLongSeq(int tag, Optional<long[]> v);
        public void writeLongSeq<T>(int tag, int count, Ice.Optional<T>
```

```
v)
            where T : IEnumerable<long>;
        public void writeLongSeq(int tag, int count, IEnumerable<long>
v);

        public void writeFloat(float v);
        public void writeFloat(int tag, float v);
        public void writeFloat(int tag, Optional<float> v);
        public void writeFloatSeq(float[] v);
        public void writeFloatSeq(int count, IEnumerable<float> v);
        public void writeFloatSeq(int tag, float[] v);
        public void writeFloatSeq(int tag, Optional<float[]> v);
        public void writeFloatSeq<T>(int tag, int count, Ice.Optional<T>
v)
            where T : IEnumerable<float>;
        public void writeFloatSeq(int tag, int count, IEnumerable<float>
v);

        public void writeDouble(double v);
        public void writeDouble(int tag, double v);
        public void writeDouble(int tag, Optional<double> v);
        public void writeDoubleSeq(double[] v);
        public void writeDoubleSeq(int count, IEnumerable<double> v);
        public void writeDoubleSeq(int tag, double[] v);
        public void writeDoubleSeq(int tag, Optional<double[]> v);
        public void writeDoubleSeq<T>(int tag, int count,
Ice.Optional<T> v)
            where T : IEnumerable<double>;
        public void writeDoubleSeq(int tag, int count,
IEnumerable<double> v);

        public void writeString(string v);
        public void writeString(int tag, string v);
        public void writeString(int tag, Optional<string> v);
        public void writeStringSeq(string[] v);
        public void writeStringSeq(int count, IEnumerable<string> v);
        public void writeStringSeq(int tag, string[] v);
        public void writeStringSeq(int tag, Optional<string[]> v);
        public void writeStringSeq<T>(int tag, int count,
Ice.Optional<T> v)
            where T : IEnumerable<string>;
        public void writeStringSeq(int tag, int count,
IEnumerable<string> v);

        public void writeSize(int sz);

        public void writeProxy(ObjectPrx v);
        public void writeProxy(int tag, ObjectPrx v);
        public void writeProxy(int tag, Optional<ObjectPrx> v);
```

```
        public void writeValue(Ice.Object v);
        public void writeValue(int tag, Ice.Object v);
        public void writeValue<T>(int tag, Ice.Optional<T> v)
            where T : Ice.Object;

        public void writeEnum(int v, int maxValue);
        public void writeEnum(int tag, int v, int maxValue);

        public void writeBlob(byte[] v);
        public void writeBlob(byte[] v, int off, int len);

        public void writeException(UserException ex);

        public void startValue(SlicedData sd);
        public void endValue();

        public void startException(SlicedData sd);
        public void endException();

        public void startSlice(String typeId, int compactId, bool last);
        public void endSlice();

        public void startEncapsulation(EncodingVersion encoding,
FormatType format);
        public void startEncapsulation();
        public void endEncapsulation();
        public void writeEmptyEncapsulation(EncodingVersion encoding);
        public void writeEncapsulation(byte[] v);

        public EncodingVersion getEncoding();

        public void writePendingValues();

        public bool writeOptional(int tag, OptionalFormat format);

        public int pos();
        public void pos(int n);

        public int startSize();
        public void endSize(int pos);

        public byte[] finished();

        public void writeSerializable(object o);

        public void resize(int sz);
```

```
            ...
        }
    }
```

Member functions are provided for inserting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void writeType(int tag, type v)`
  `void writeType(int tag, Optional<type> v)`
  `void writeTypeSeq(int tag, type[] v)`
  `void writeTypeSeq(int tag, Optional<type[]> v)`
  `void writeTypeSeq(int tag, int count, IEnumerable<type[]> v)`
  Inserts an optional value. Methods that accept `Optional` instances only insert a value if the argument is non-null and contains a value.

- `void rewriteByte(byte v, int dest)`
  `void rewriteBool(bool v, int dest)`
  `void rewriteInt(int v, int dest)`
  Overwrites the byte(s) at an existing location in the buffer at the given destination with a value. These methods do not change the current position of the stream.

- `void writeSize(int sz)`
  The Ice encoding has a compact representation to indicate size. This function converts the given non-negative integer into the proper encoded representation.

- `void writeProxy(Ice.ObjectPrx v)`
  Inserts a proxy.

- `void writeValue(Ice.Object v)`
  Inserts an instance of a Slice class. The Ice encoding for class instances may cause the insertion to be delayed, in which case the stream retains a reference to the given instance and does not insert its state it until `writePendingValues` is invoked on the stream.

- `void writeEnum(int val, int maxValue)`
  Writes the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.

- `void writeBlob(byte[] v)`
  `void writeBlob(byte[] v, int off, int len)`
  Copies the given blob of bytes directly to the stream's internal buffer without modification.

- `void writeException(UserException ex)`
  Inserts a user exception.

- `void startValue(SlicedData sd)`
  `void endValue()`
  When marshaling the slices of a class instance, the application must first call `startValue`, then marshal the slices, and finally call `endValue`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or null if there are no preserved slices.

- `void startException(SlicedData sd)`
  `void endException()`
  When marshaling the slices of an exception, the application must first call `startException`, then marshal the slices, and finally call `endException`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or 0 if there are no preserved slices.

- `void startSlice(String typeId, int compactId, bool last)`
  `void endSlice()`
  Starts and ends a slice of class or exception member data. The call to `startSlice` must include the type ID for the current slice, the corresponding compact ID for the type (if any), and a boolean indicating whether this is the last slice of the class instance or exception. The compact ID is only relevant for class instances; pass a negative value to indicate the encoding should use the string type ID.

- `void startEncapsulation(EncodingVersion encoding, FormatType format)`
  `void startEncapsulation()`
  `void endEncapsulation()`
  Starts and ends an encapsulation, respectively. The first overloading of `startEncapsulation` allows you to specify the encoding version as well as the format to use for any class instances and exceptions marshaled within this encapsulation.

- `void writeEmptyEncapsulation(EncodingVersion encoding)`
  Writes an encapsulation having the given encoding version with no encoded data.

- `void writeEncapsulation(byte[] v)`
  Copies the bytes representing an encapsulation from the given array into the stream.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently being used by the stream.

- `void writePendingValues()`
  Encodes the state of class instances whose insertion was delayed during `writeValue`. This member function must only be called once. For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `bool writeOptional(int tag, OptionalFormat fmt)`
  Prepares the stream to write an optional value with the given tag and format. Returns true if the value should be written, or false otherwise. A return value of false indicates that the encoding version in use by the stream does not support optional values. If this method returns true, the data associated with that optional value must be written next. Optional values must be written in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  C#

  ```
  namespace Ice
  {
      public enum OptionalFormat
      {
          OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
  OptionalFormatF8,
          OptionalFormatSize, OptionalFormatVSize,
  OptionalFormatFSize,
          OptionalFormatEndMarker
      }
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.

- `int pos()`
  `void pos(int n)`
  Returns or changes the stream's current position, respectively.

- `int startSize()`
  `void endSize(int n)`
  The encoding for optional values uses a 32-bit integer to hold the size of variable-length types. Calling `startSize` writes a placeholder value for the size and returns the starting position of the size value; after writing the data, call `endSize` to patch the placeholder with the actual size at the given position.

- `byte[] finished()`
  Indicates that marshaling is complete and returns the encoded byte sequence. This member function must only be called once.

- `void writeSerializable(object v)`
  Writes a serializable object to the stream.
- `void resize(int sz)`
  Allocates space for `sz` more bytes in the buffer. The stream implementation internally uses this function prior to copying more data into the buffer.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in C-Sharp

Stream Helper Methods in C-Sharp

The stream classes provide all of the low-level methods necessary for encoding and decoding Ice types. However, it would be tedious and error-prone to manually encode complex Ice types such as classes, structs, and dictionaries using these low-level functions. For this reason, the Slice compiler optionally generates helper methods for streaming complex Ice types.

We will use the following Slice definitions to demonstrate the language mapping:

Slice

```
module M
{
    sequence<...> Seq;
    dictionary<...> Dict;
    struct S
    {
        ...
    }
    enum E { ... }
    class C
    {
        ...
    }
    interface I
    {
        ...
    }
}
```

The Slice compiler generates the corresponding helper methods shown below:

C#

```
namespace M
{
    public sealed class SeqHelper
    {
        public static ...[] read(Ice.InputStream istr);
        public static void write(Ice.OutputStream ostr, ...[] v);
    }

    public sealed class DictHelper
    {
        public static Dictionary<...> read(Ice.InputStream istr);
        public static void write(Ice.OutputStream ostr, Dictionary<...>
v);
    }

    public partial struct S
    {
        public static S ice_read(Ice.InputStream istr);
        public static void ice_write(Ice.OutputStream ostr, S v);
        ...
        // Instance methods
        public void ice_read(Ice.InputStream istr);
        public void ice_write(Ice.OutputStream ostr);
    }

    public sealed class EHelper
    {
        public static E read(Ice.InputStream istr);
        public static void write(Ice.OutputStream ostr, E v);
    }

    public sealed class IPrxHelper : ...
    {
        public static IPrx read(Ice.InputStream istr);
        public static void write(Ice.OutputStream ostr, IPrx v);
        ...
    }
}
```

The `IPrxHelper` class provides `read` and `write` methods for extracting and inserting proxies, respectively. Note that the `read` method returns a proxy of type `IPrx` but does not perform the equivalent of a `checkedCast` to verify that the remote object supports interface `I`.

No additional code is generated for marshaling instances of class types, such as type `C` that we defined above. Applications should use `OutputStream.writeValue` and `InputStream.readValue` to insert and extract class instances, respectively.

See Also

- slice2cs Command-Line Options
- The InputStream Interface in C-Sharp
- The OutputStream Interface in C-Sharp

JavaScript Streaming Interfaces

The stream API allows you to manually marshal and unmarshal Slice types using the Ice data encoding.

Topics

The InputStream Interface in JavaScript

On this page:

Initializing an `InputStream` in JavaScript

The `InputStream` class provides a constructor that accepts up to three arguments. Legal argument values are:

- A communicator instance
- An encoding version
- A byte array containing the encoded data that you intend to decode

You'll normally supply the encoded data argument. The stream does not make a copy of this data; rather, it uses the data as supplied and assumes it remains unmodified for the lifetime of the stream object.

We recommend supplying a communicator instance, otherwise you will not be able to decode proxy objects. The stream also inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

Use the following properties to manually configure the stream:

JavaScript

```
var stream = new Ice.InputStream(...);
stream.valueFactoryManager = ...
stream.logger = ...
stream.compactIdResolver = ...
stream.sliceValues = ...
stream.traceSlicing = ...
```

The settings include:

- Value factory manager
  A value factory manager supplies custom factories for Slice class types.
- Logger
  The stream uses a logger to record warning and trace messages.
- Compact ID resolver
  A compact ID resolver for translating numeric values into Slice type IDs. The stream invokes the resolver by passing it the numeric compact ID. The resolver is expected to return the Slice type ID associated with that numeric ID or an empty string if the numeric ID is unknown. The application must supply a function that accepts an integer and returns a string.
- Slice values
  The flag indicates whether to slice instances of Slice classes to a known Slice type when a more derived type is unknown. An instance is "sliced" when no static information is available for a Slice type ID and no factory can be found for that type, resulting in the creation of an instance of a less-derived type. If slicing is disabled in this situation, the stream raises the exception `NoValueFactoryException`. The default behavior is to allow slicing.
- Trace slicing
  The flag indicates whether to log messages when instances of Slice classes are sliced. If the stream is initialized with a communicator, this setting defaults to the value of the `Ice.Trace.Slicing` property, otherwise the setting defaults to false.

Extracting from an `InputStream` in JavaScript

`InputStream` provides a number of `read` methods that allow you to extract Slice types from the stream.

For example, you can extract a boolean and a string from a stream as follows:

JavaScript

```
var data = ...
var istr = new Ice.InputStream(communicator, data);
var b = istr.readBool();
var s = istr.readString();
```

Here are the methods for extracting data from a stream:

- `readByte()`
  `readBool()`
  `readInt()`
  `readLong()`
  `readFloat()`
  `readDouble()`
  `readString()`
  Extracts and returns a primitive value.
- `readByteSeq()`
  Extracts and returns a byte sequence as a "slice" of the internal buffer.
- `readSize()`
  The Ice encoding has a compact representation to indicate size. This function extracts a size and returns it as an integer.
- `readAndCheckSeqSize(minWireSize)`
  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element. If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.
- `readProxy(type)`
  If no argument is provided, this function returns an instance of the base proxy type, `ObjectPrx`, otherwise it returns an instance of the given proxy type.
- `readValue(cb, type)`
  The Ice encoding for class instances requires extraction to occur in stages. The `readValue` function accepts a callback function that receives the extracted instance as its argument. The caller must also supply the class type expected for the instance. When the instance is available, the callback function is called. The application must call `readPendingValues` on the stream to ensure that all instances are properly extracted.
- `readEnum(type)`
  Extracts the integer value of an enumerator and returns the equivalent enumerator. The caller must pass the enumeration type as the argument.
- `throwException()`
  Extracts a user exception from the stream and throws it. If the stored exception is of an unknown type, the function attempts to extract and throw a less-derived exception. If that also fails, an exception is thrown: for the 1.0 encoding, the exception is `UnmarshalOutOfBoundsException`, for the 1.1 encoding, the exception is `UnknownUserException`.
- `startSlice()`
  `endSlice()`
  `skipSlice()`
  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of a class instance or user exception. The `startSlice` method returns the type ID of the next slice, which may be an empty string depending on the format used to encode the instance or exception.
- `startValue()`
  `endValue(preserve)`
  The `startValue` method must be called prior to reading the slices of a class instance. The `endValue` method must be called after all slices have been read. Pass true to `endValue` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endValue` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the instance with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `startException()`
  `endException(preserve)`
  The `startException` method must be called prior to reading the slices of an exception. The `endException` method must be called after all slices have been read. Pass true to `endException` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endException` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the exception with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `startEncapsulation()`
  `endEncapsulation()`
  `skipEncapsulation()`
  Start, end, and skip an encapsulation, respectively. The `startEncapsulation` and `skipEncapsulation` methods return the encoding version used to encode the contents of the encapsulation.

- `skipEmptyEncapsulation(version)`
  Skips an encapsulation that is expected to be empty. The stream raises `EncapsulationException` if the encapsulation is not empty. If the `version` argument is non-null, the stream stores the encoding version of the empty encapsulation in `version`.

- `getEncapsulationSize()`
  Returns the size of the current encapsulation.

- `getEncoding()`
  Returns the encoding version currently in use by the stream.

- `readPendingValues()`
  An application must call this function after all other data has been extracted, but only if class instances were encoded. This function extracts the state of class instances and invokes their corresponding callback functions (see `readValue`). For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `readBlob(sz)`
  Extracts an array of `sz` bytes from the stream at its current position. The returned byte array contains undecoded data from the stream's internal buffer. The stream's position is advanced by `sz`.

- `skip(sz)`
  Skips the given number of bytes.

- `skipSize()`
  Reads a size at the current position and skips that number of bytes.

- `readOptional(tag, fmt)`
  Returns true if an optional value with the given tag and format is present, or false otherwise. If this method returns true, the data associated with that optional value must be read next. Optional values must be read in order by tag from least to greatest. The `OptionalFormat` enumeration is equivalent to the following:

  Slice

  ```
  module Ice
  {
      enum OptionalFormat
      {
          OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
  OptionalFormatF8,
          OptionalFormatSize, OptionalFormatVSize,
  OptionalFormatFSize,
          OptionalFormatEndMarker
      }
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.

- `readOptionalHelper(tag, fmt, readFn)`

If an optional value is present with the given tag and format, the stream invokes the given helper function to extract the value. For example, you can pass `InputStream.prototype.readByte` as the helper function when extracting an optional byte value. This function returns `undefined` if the optional value is not present.

- `readOptionalProxy(tag, type)`
  If an optional value is present with the given tag and the correct format for a proxy, this function extracts the proxy information and returns a proxy instance of the given type, or `ObjectPrx` if no type is provided. This function returns `undefined` if the optional value is not present.

- `readOptionalEnum(tag, type)`
  If an optional value is present with the given tag and the correct format for the given enumeration type, this function extracts the enumerator and returns it. This function returns `undefined` if the optional value is not present.

- `readOptionalValue(tag, cb, type)`
  If an optional class instance is present with the given tag and the correct format for a class instance, this function begins the process of extracting the instance. The stream will invoke the callback function and pass it the instance as an argument. The caller must also supply the class type expected for the instance. The callback function receives the value `undefined` if the optional instance is not present.

Finally, you can use the `pos` property to get and set the stream's current position.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions

The OutputStream Interface in JavaScript

On this page:

Initializing an `OutputStream` in JavaScript

The `OutputStream` class provides a constructor that accepts up to two arguments:

- A communicator instance
- An encoding version

We recommend supplying a communicator instance. The stream inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually as we describe below.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

Use the following property to manually configure the stream:

JavaScript

```
var stream = new Ice.OutputStream(...);
stream.format = ...;
```

For instances of Slice classes, the format determines how the slices of an instance are encoded. If the stream is initialized with a communicator, this setting defaults to the value of `Ice.Default.SlicedFormat`, otherwise the setting defaults to the compact format.

Inserting into an `OutputStream` in JavaScript

`OutputStream` provides a number of `write` methods that allow you to insert Slice types into the stream.

For example, you can insert a boolean and a string into a stream as follows:

Java

```
Ice.OutputStream out = new Ice.OutputStream(communicator);
out.writeBool(true);
out.writeString("Ice rocks!");
var data = out.finished();
```

Here are the methods for inserting data into a stream:

- `writeByte(v)`
  `writeBool(v)`
  `writeShort(v)`
  `writeInt(v)`
  `writeLong(v)`
  `writeFloat(v)`
  `writeDouble(v)`
  `writeString(v)`
  Inserts a primitive value.
- `writeByteSeq(v)`
  Inserts an array as a byte sequence.
- `rewriteByte(v, dest)`
  `rewriteBool(v, dest)`
  `rewriteInt(v, dest)`

Overwrites the byte(s) at an existing location in the buffer at the given destination with a value. These methods do not change the current position of the stream.

- `writeSize(sz)`
  The Ice encoding has a compact representation to indicate size. This function converts the given non-negative integer into the proper encoded representation.

- `writeProxy(v)`
  `writeOptionalProxy(tag, v)`
  Inserts a proxy or an optional proxy.

- `writeValue(v)`
  `writeOptionalValue(tag, v)`
  Inserts an instance of a Slice class or an optional instance. The Ice encoding for class instances may cause the insertion to be delayed, in which case the stream retains a reference to the given instance and does not insert its state it until `writePendingValues` is invoked on the stream.

- `writeEnum(v)`
  Writes the integer value of an enumerator.

- `writeBlob(v)`
  Copies the given array of bytes directly to the stream's internal buffer without modification.

- `writeException(ex)`
  Inserts a user exception.

- `startValue(slicedData)`
  `endValue()`
  When marshaling the slices of a class instance, the application must first call `startValue`, then marshal the slices, and finally call `endValue`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or null if there are no preserved slices.

- `startException(slicedData)`
  `endException()`
  When marshaling the slices of an exception, the application must first call `startException`, then marshal the slices, and finally call `endException`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or 0 if there are no preserved slices.

- `startSlice(typeId, compactId, last)`
  `endSlice()`
  Starts and ends a slice of class or exception member data. The call to `startSlice` must include the type ID for the current slice, the corresponding compact ID for the type (if any), and a boolean indicating whether this is the last slice of the class instance or exception. The compact ID is only relevant for class instances; pass a negative value to indicate the encoding should use the string type ID.

- `startEncapsulation(encoding, fmt)`
  `endEncapsulation()`
  Starts and ends an encapsulation, respectively. When calling `startEncapsulation`, you can optionally specify the encoding version as well as the format to use for any class instances and exceptions marshaled within this encapsulation.

- `writeEmptyEncapsulation(encoding)`
  Writes an encapsulation having the given encoding version with no encoded data.

- `writeEncapsulation(v)`
  Copies the bytes representing an encapsulation from the given array into the stream.

- `getEncoding()`
  Returns the encoding version currently being used by the stream.

- `writePendingValues()`
  Encodes the state of class instances whose insertion was delayed during `writeValue`. This member function must only be called once. For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `writeOptional(tag, fmt)`
  Prepares the stream to write an optional value with the given tag and format. Returns true if the value should be written, or false otherwise. A return value of false indicates that the encoding version in use by the stream does not support optional values. If this method returns true, the data associated with that optional value must be written next. Optional values must be written in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

Slice

```
module Ice
{
    enum OptionalFormat
    {
        OptionalFormatF1, OptionalFormatF2, OptionalFormatF4,
OptionalFormatF8,
        OptionalFormatSize, OptionalFormatVSize,
OptionalFormatFSize,
        OptionalFormatEndMarker
    }
}
```

Refer to the encoding discussion for more information on the meaning of these values.

- `writeOptionalHelper(tag, fmt, writeFn, v)`
  If the value `v` is not `undefined`, this method writes an optional value using the given tag and format. The helper function `writeFn` is called to insert the value in `v`. For example, you can pass `OutputStream.prototype.writeByte` as the helper function when writing an optional byte.

- `startSize()`
  `endSize(n)`
  The encoding for optional values uses a 32-bit integer to hold the size of variable-length types. Calling `startSize` writes a placeholder value for the size and returns the starting position of the size value; after writing the data, call `endSize` to patch the placeholder with the actual size at the given position.

- `finished()`
  Indicates that marshaling is complete and returns the encoded byte sequence as an array. This member function must only be called once.

- `resize(sz)`
  Allocates space for `sz` more bytes in the buffer. The stream implementation internally uses this function prior to copying more data into the buffer.

Finally, you can use the `pos` property to get and set the stream's current position.

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions

Stream Helper Methods in JavaScript

The stream classes provide all of the low-level methods necessary for encoding and decoding Ice types. However, it would be tedious and error-prone to manually encode complex Ice types such as classes, structs, and dictionaries using these low-level functions. For this reason, the Slice compiler generates helper methods for streaming user-defined types.

Sequence

For a Slice sequence named `StringSeq`, the Slice compiler generates `StringSeqHelper` that provides `read` and `write` methods:

JavaScript

```
var v = StringSeqHelper.read(istr);
StringSeqHelper.write(ostr, v);
```

Dictionary

For a Slice dictionary named `EmployeeMap`, the Slice compiler generates `EmployeeMapHelper` that provides `read` and `write` methods:

JavaScript

```
var v = EmployeeMapHelper.read(istr);
EmployeeMapHelper.write(ostr, v);
```

Enumeration

For a Slice enumeration named `Color`, the Slice compiler generates the JavaScript class `Color`. No additional helper methods are necessary because the stream classes make insertion and extraction of enumerators straightforward:

JavaScript

```
var e = istr.readEnum(Color);
ostr.writeEnum(e);
```

Structure

For a Slice structure named `Point`, the Slice compiler generates the JavaScript class `Point`. This class defines "static" `read` and `write` methods:

JavaScript

```
var p = Point.read(istr);
Point.write(ostr, p);
```

Class

For a Slice class named `Shape`, the Slice compiler generates the JavaScript class `Shape`. This class defines "static" `read` and `write` methods:

JavaScript

```
var obj = Shape.read(istr); // obj.value holds instance
istr.readPendingValues();
Shape.write(ostr, obj.value);
```

Note that `read` returns a wrapper object with a `value` member. This member may not be initialized until after `readPendingValues` has been called on the stream and all instances have been unmarshaled, at which point `obj.value` will either be `null` or a reference to an instance.

The generated class also provides methods for handling optional instances:

JavaScript

```
var optional = Shape.readOptional(istr, tag);
istr.readPendingValues();
if(optional.value !== undefined)
{
    Shape.writeOptional(ostr, tag, optional.value);
}
```

Again, the return value of `readOptional` is a wrapper whose `value` member will eventually be set to `undefined` (if the optional instance was not present), `null`, or a reference to an instance.

See Also

- Data Encoding
- slice2java Command-Line Options

Dynamic Invocation and Dispatch

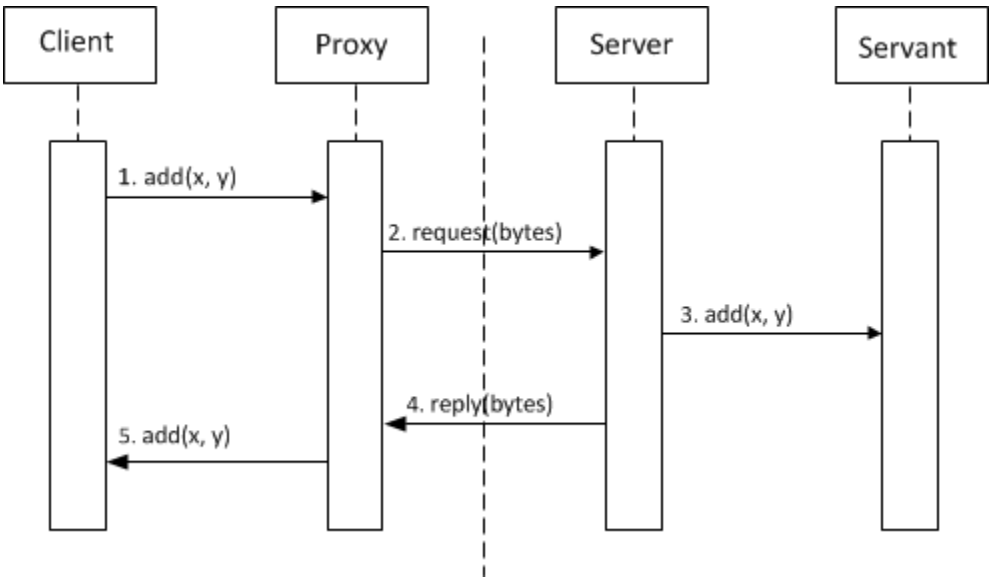
Topics

Dynamic Invocation and Dispatch Overview

On this page:

Use Cases for Dynamic Invocation and Dispatch

Ice applications generally use the static invocation model, in which the application invokes a Slice operation by calling a member function on a generated proxy class. In the server, the static dispatch model behaves similarly: the request is dispatched to the servant as a statically-typed call to a member function. Underneath this statically-typed facade, the Ice run times in the client and server are exchanging sequences of bytes representing the encoded request arguments and results. These interactions are illustrated below:



Interactions in a static invocation.

1. The client initiates a call to the Slice operation `add` by calling the member function `add` on a proxy.
2. The generated proxy class marshals the arguments into a sequence of bytes and transmits them to the server.
3. In the server, the generated servant class unmarshals the arguments and calls `add` on the subclass.
4. The servant marshals the results and returns them to the client.
5. Finally, the client's proxy unmarshals the results and returns them to the caller.

The application is blissfully unaware of this low-level machinery, and in the majority of cases that is a distinct advantage. In some situations, however, an application can leverage this machinery to accomplish tasks that are not possible in a statically-typed environment. Ice provides the dynamic invocation and dispatch models for these situations, allowing applications to send and receive requests as encoded sequences of bytes instead of statically-typed arguments.

The dynamic invocation and dispatch models offer several unique advantages to Ice services that forward requests from senders to receivers, such as Glacier2 and IceStorm. For these services, the request arguments are an opaque byte sequence that can be forwarded without the need to unmarshal and remarshal the arguments. Not only is this significantly more efficient than a statically-typed implementation, it also allows intermediaries such as Glacier2 and IceStorm to be ignorant of the Slice types in use by senders and receivers.

Another use case for the dynamic invocation and dispatch models is scripting language integration. The Ice extensions for Python, PHP, and Ruby invoke Slice operations using the dynamic invocation model; the request arguments are encoded using the streaming interfaces.

It may be difficult to resist the temptation of using a feature like dynamic invocation or dispatch, but we recommend that you carefully consider the risks and complexities of such a decision. For example, an application that uses the streaming interface to manually encode and decode request arguments has a high risk of failure if the argument signature of an operation changes. In contrast, this risk is greatly reduced in the static invocation and dispatch models because errors in a strongly-typed language are found early, during compilation. Therefore, we caution you against using this capability except where its advantages significantly outweigh the risks.

Dynamic Invocation using `ice_invoke`

Dynamic invocation is performed using the proxy member function `ice_invoke`, defined in the proxy base class `ObjectPrx`. If we were to define this operation in Slice, it would look like this:

Slice

```
sequence<byte> ByteSeq;

bool ice_invoke(string operation,
                Ice::OperationMode mode,
                ByteSeq inParams,
                out ByteSeq outParams);
```

The first argument is the name of the Slice operation.

> This is the Slice name of the operation, not the name as it might be mapped to any particular language. For example, the string `"while"` is the name of the Slice operation `while`, and not `"_cpp_while"` (C++) or `"_while"` (Java).

The second argument is an enumerator from the Slice type `Ice::OperationMode`; the possible values are `Normal` and `Idempotent`. The third argument, `inParams`, represents an encapsulation of the encoded in-parameters of the operation.

A return value of `true` indicates a successful invocation, in which case an encapsulation of the marshaled form of the operation's results (if any) is provided in `outParams`. A return value of `false` signals the occurrence of a user exception whose encapsulated data is provided in `outParams`. The caller must also be prepared to catch local exceptions, which are thrown directly.

Dynamic Dispatch using `Blobject`

A server enables dynamic dispatch by creating a subclass of `Blobject` (the name is derived from *blob*, meaning a blob of bytes). The Slice equivalent of `Blobject` is shown below:

Slice

```
sequence<byte> ByteSeq;

interface Blobject
{
    bool ice_invoke(ByteSeq inParams, out ByteSeq outParams);
}
```

The `inParams` argument supplies an encapsulation of the encoded in-parameters. The contents of the `outParams` argument depends on the outcome of the invocation: if the operation succeeded, `ice_invoke` must return `true` and place an encapsulation of the encoded results in `outParams`; if a user exception occurred, `ice_invoke` must return `false`, in which case `outParams` contains an encapsulation of the encoded exception. The operation may also raise local exceptions such as `OperationNotExistException`.

The language mappings add a trailing argument of type `Ice::Current` to `ice_invoke`, and this provides the implementation with the name of the operation being dispatched.

Because `Blobject` derives from `Object`, an instance is a regular Ice servant just like instances of the classes generated for user-defined Slice interfaces. The primary difference is that all operation invocations on a `Blobject` instance are dispatched through the `ice_invoke` m ember function.

If a `Blobject` subclass intends to decode the in-parameters (and not simply forward the request to another object), then the implementation obviously must know the signatures of all operations it supports. How a `Blobject` subclass determines its type information is an implementation detail that is beyond the scope of this manual.

Note that a `Blobject` servant is also useful if you want to create a message forwarding service, such as Glacier2. In this case, there is no need to decode any parameters; instead, the implementation simply forwards each request unchanged to a new destination. You can register a `Blobject` servant as a default servant to easily achieve this.

See Also

- Collocated Invocation and Dispatch
- The Current Object
- Default Servants
- Streaming Interfaces
- Glacier2
- IceStorm

Dynamic Invocation

This page describes the language mappings for the `ice_invoke` proxy function (on `ObjectPrx`).

On this page:

Synchronous Mapping for `ice_invoke`

The mapping for `ice_invoke` follows the standard rules for each language mapping.

`ice_invoke` API

The synchronous mapping for `ice_invoke` on `ObjectPrx` is shown below:

C++11 C++98 C# Java Java Compat ObjC Python

```
bool ice_invoke(const std::string& operation,
                Ice::OperationMode mode,
                const std::vector<Ice::Byte>& inParams,
                std::vector<Ice::Byte>& outParams,
                const Ice::Context& context = Ice::noExplicitContext);

// array-mapping overload that maps the sequence<byte> input parameter
to an array:
//
bool ice_invoke(const std::string& operation,
                Ice::OperationMode mode,
                const std::pair<const Ice::Byte*, const Ice::Byte*>& in,
                std::vector<Ice::Byte>& out,
                const Ice::Context& = Ice::noExplicitContext);
```

```
bool ice_invoke(const std::string& operation,
                Ice::OperationMode mode,
                const std::vector<Ice::Byte>& inParams,
                std::vector<Ice::Byte>& outParams,
                const Ice::Context& context = Ice::noExplicitContext);

// array-mapping overload that maps the sequence<byte> input parameter
to an array:
//
bool ice_invoke(const std::string& operation,
                Ice::OperationMode mode,
                const std::pair<const Ice::Byte*, const Ice::Byte*>& in,
                std::vector<Ice::Byte>& out,
                const Ice::Context& = Ice::noExplicitContext);
```

```
namespace Ice
{
    public interface ObjectPrx
    {
        bool ice_invoke(string operation,
                        OperationMode mode,
                        byte[] inParams,
                        out byte[] outParams);
        // ...
    }
}
```

Another overload of `ice_invoke` (not shown) adds a trailing argument of type `Ice.Context`.

```
package com.zeroc.Ice;

public interface Object
{
    public class Ice_invokeResult
    {
        public Ice_invokeResult()
        {
        }

        public Ice_invokeResult(boolean returnValue, byte[] outParams)
        {
            this.returnValue = returnValue;
            this.outParams = outParams;
        }

        public boolean returnValue;
        public byte[] outParams;
    }

    ...
}

public interface ObjectPrx
{
    com.zeroc.Ice.Object.Ice_invokeResult ice_invoke(String operation,
OperationMode mode, byte[] inParams);

    ...
}
```

Another overload of `ice_invoke` (not shown) adds a trailing argument of type `Ice.Context`.

```
boolean ice_invoke(String operation,
                   Ice.OperationMode mode,
                   byte[] inParams,
                   Ice.ByteSeqHolder outParams);
```

Another overload of `ice_invoke` (not shown) adds a trailing argument of type `Ice.Context`.

```
-(BOOL) ice_invoke:(NSString*)operation mode:(ICEOperationMode)mode
inEncaps:(NSData*)inEncaps outEncaps:(NSMutableData**)outEncaps;
```

Another overload of `ice_invoke` (not shown) adds a trailing argument of type `Ice.Context`.

```
def ice_invoke(self, operation, mode, inParams, context=None)
```

Upon completion, the method returns a tuple consisting of `(ok, outParams)`, where `ok` is `True` if the invocation completed successfully and `outParams` contains the encapsulated output parameters, or `False` if the invocation resulted in a user exception and `outParams` cont ains the encapsulated exception.

Calling `ice_invoke`

The code below demonstrates how to invoke the operation `op`, which takes no in parameters:

C++11C++98C#JavaJava CompatPython

```
std::shared_ptr<Ice::ObjectPrx> proxy = ...
try
{
    std::vector<Ice::Byte> inParams, outParams;
    if(proxy->ice_invoke("op", Ice::OperationMode::Normal, inParams,
outParams))
    {
        // Handle success
    }
    else
    {
        // Handle user exception
    }
}
catch(const Ice::LocalException& ex)
{
    // Handle exception
}
```

As a convenience, the Ice run time accepts an empty byte sequence when there are no input parameters and internally translates it into an empty encapsulation. In all other cases, the value for `inParams` must be an encapsulation of the encoded parameters.

```
Ice::ObjectPrx proxy = ...
try
{
    std::vector<Ice::Byte> inParams, outParams;
    if(proxy->ice_invoke("op", Ice::Normal, inParams, outParams))
    {
        // Handle success
    }
    else
    {
        // Handle user exception
    }
}
catch(const Ice::LocalException& ex)
{
    // Handle exception
}
```

As a convenience, the Ice run time accepts an empty byte sequence when there are no input parameters and internally translates it into an empty encapsulation. In all other cases, the value for `inParams` must be an encapsulation of the encoded parameters.

```
Ice.ObjectPrx proxy = ...
try
{
    byte[] outParams;
    if(proxy.ice_invoke("op", Ice.OperationMode.Normal, null, out
outParams))
    {
        // Handle success
    }
    else
    {
        // Handle user exception
    }
}
catch (Ice.LocalException ex)
{
    // Handle exception
}
```

As a convenience, the Ice run time accepts an empty or null byte sequence when there are no input parameters and internally translates it into an empty encapsulation. In all other cases, the value for `inParams` must be an encapsulation of the encoded parameters.

```
com.zeroc.Ice.ObjectPrx proxy = ...
try
{
    com.zeroc.Ice.Object.Ice_invokeResult r = proxy.ice_invoke("op",
com.zeroc.Ice.OperationMode.Normal, null);
    if(r.returnValue)
    {
        // Handle success
    }
    else
    {
        // Handle user exception
    }
}
catch(com.zeroc.Ice.LocalException ex)
{
    // Handle exception
}
```

As a convenience, the Ice run time accepts an empty or null byte sequence when there are no input parameters and internally translates it into an empty encapsulation. In all other cases, the value for `inParams` must be an encapsulation of the encoded parameters.

```
Ice.ObjectPrx proxy = ...
try
{
    Ice.ByteSeqHolder outParams = new Ice.ByteSeqHolder();
    if(proxy.ice_invoke("op", Ice.OperationMode.Normal, null,
outParams))
    {
        // Handle success
    }
    else
    {
        // Handle user exception
    }
}
catch(Ice.LocalException ex)
{
    // Handle exception
}
```

As a convenience, the Ice run time accepts an empty or null byte sequence when there are no input parameters and internally translates it into an empty encapsulation. In all other cases, the value for `inParams` must be an encapsulation of the encoded parameters.

```
proxy = ...
try:
    [ok, outParams] = proxy.ice_invoke("op", Ice.OperationMode.Normal,
None)
    if ok:
        # Handle success
    else:
        # Handle user exception
except Ice.LocalException as ex:
    # Handle exception
```

Asynchronous Mapping for `ice_invoke`

The asynchronous mapping for `ice_invoke` resembles the static AMI mapping. The return value and the parameters `operation`, `mode`, and `inParams` have the same semantics as for the synchronous version of `ice_invoke` shown above.

C++11C++98C# AsyncResult APIC# Task APIJavaJava CompatObjCPython

Proxy member functions

```
// "future" async function
// with P = std::promise (the default), it returns a
std::future<Ice::Object::Ice_invokeResult>
//
template<template<typename> class P = std::promise> auto
ice_invokeAsync(const std::string& operation,
                Ice::OperationMode mode,
                const std::vector<Byte>& inP,
                const ::Ice::Context& context = Ice::noExplicitContext)
        ->
decltype(std::declval<P<Ice::Object::Ice_invokeResult>>().get_future());

// async function with callbacks
//
std::function<void()>
ice_invokeAsync(const std::string& operation,
                Ice::OperationMode mode,
                const std::vector<Ice::Byte>& inP,
                std::function<void(bool, const std::vector<Ice::Byte>&)>
response,
                std::function<void(std::exception_ptr)> ex = nullptr,
                std::function<void(bool)> sent = nullptr,
                const Ice::Context& context = Ice::noExplicitContext);

// "future" async function, with "zero-copy" in parameter
// with P = std::promise (the default), it returns a
std::future<Ice::Object::Ice_invokeResult>
//
template<template<typename> class P = std::promise> auto
ice_invokeAsync(const std::string& operation,
                Ice::OperationMode mode,
                const std::pair<const Ice::Byte*, const Ice::Byte*>&
inP,
                const Ice::Context& context = Ice::noExplicitContext)
        ->
decltype(std::declval<P<Ice::Object::Ice_invokeResult>>().get_future());


// async function with callbacks, and "zero-copy" in parameter
//
std::function<void()>
ice_invokeAsync(const std::string& operation,
                Ice::OperationMode mode,
                const std::pair<const Ice::Byte*, const Ice::Byte*>&
inP,
                std::function<void(bool, const std::pair<const
Ice::Byte*, const Ice::Byte*>&)> response,
                std::function<void(std::exception_ptr)> ex = nullptr,
                std::function<void(bool)> sent = nullptr,
                const Ice::Context& context = Ice::noExplicitContext);
```

Ice::Object

```
class Object
{
public:
    ...
    struct Ice_invokeResult
    {
        bool returnValue;
        std::vector<Ice::Byte> outParams;
    };
};
```

Proxy member functions

```
// Basic (no callback)
//
Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::vector<Ice::Byte>&
inParams);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::vector<Ice::Byte>&
inParams,
                                     const Ice::Context& context);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::pair<const Ice::Byte*,
const Ice::Byte*>& inParams);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::pair<const Ice::Byte*,
const Ice::Byte*>& inParams,
                                     const Ice::Context& context,
                                     const Ice::LocalObjectPtr& cookie
= 0);

bool end_ice_invoke(std::vector<Ice::Byte>&, const
Ice::AsyncResultPtr&);

// With generic callback - use Ice::newCallback to create these generic
callbacks
//
```

```
Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::vector<Ice::Byte>&
inParams,
                                     const:Ice::CallbackPtr& del,
                                     const Ice::LocalObjectPtr& cookie
= 0);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::vector<Ice::Byte>&
inParams,
                                     const Ice::Context& context,
                                     const Ice::CallbackPtr& del,
                                     const Ice::LocalObjectPtr& cookie
= 0);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::pair<const Ice::Byte*,
const Ice::Byte*>& inParams,
                                     const Ice::CallbackPtr& del,
                                     const Ice::LocalObjectPtr& cookie
= 0);


Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::pair<const Ice::Byte*,
const Ice::Byte*>& inParams,
                                     const Ice::Context& context,
                                     const Ice::CallbackPtr& del,
                                     const Ice::LocalObjectPtr& cookie
= 0);

// With type-safe callback - use newCallback_Object_ice_invoke (below)
to create these callbacks
//
Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::vector<Ice::Byte>&
inParams,
                                     const
Ice::Callback_Object_ice_invokePtr& del,
                                     const Ice::LocalObjectPtr& cookie
= 0);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                     Ice::OperationMode mode,
                                     const std::vector<Ice::Byte>&
```

```
inParams,
                                    const Ice::Context& context,
                                    const
Ice::Callback_Object_ice_invokePtr& del,
                                    const Ice::LocalObjectPtr& cookie
= 0);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                    Ice::OperationMode mode,
                                    const std::pair<const Ice::Byte*,
const Ice::Byte*>& inParams,
                                    const
Ice::Callback_Object_ice_invokePtr& del,
                                    const Ice::LocalObjectPtr& cookie
= 0);

Ice::AsyncResultPtr begin_ice_invoke(const std::string& op,
                                    Ice::OperationMode mode,
                                    const std::pair<const Ice::Byte*,
const Ice::Byte*>& inParams,
                                    const Ice::Context& context,
                                    const
Ice::Callback_Object_ice_invokePtr& del,
                                    const Ice::LocalObjectPtr& cookie
= 0);

// factory functions for type-safe callbacks
// Versions are provided to support zero-copy semantics for the byte
sequence containing the operation's results,
// as well as a cookie value whose type is inferred and represented here
by the CT (cookie type) symbol
//
template<class T> Callback_Object_ice_invokePtr
newCallback_Object_ice_invoke(const IceUtil::Handle<T>& instance,
                              void (T::*cb)(bool, const
std::vector<Ice::Byte>&),
                              void (T::*excb)(const Ice::Exception&),
                              void (T::*sentcb)(bool) = 0);

template<class T> Callback_Object_ice_invokePtr
newCallback_Object_ice_invoke(const IceUtil::Handle<T>& instance,
                              void (T::*cb)(bool, const std::pair<const
Ice::Byte*, const Ice::Byte*>&),
                              void (T::*excb)(const Ice::Exception&),
                              void (T::*sentcb)(bool) = 0);

template<class T, typename CT> Callback_Object_ice_invokePtr
newCallback_Object_ice_invoke(const IceUtil::Handle<T>& instance,
                              void (T::*cb)(bool, const
std::vector<Ice::Byte>&, const CT&),
```

```
                                void (T::*excb)(const Ice::Exception&,
const CT&),
                                void (T::*sentcb)(bool, const CT&) = 0);

template<class T, typename CT> Callback_Object_ice_invokePtr
newCallback_Object_ice_invoke(const IceUtil::Handle<T>& instance,
                                void (T::*cb)(bool, const std::pair<const
Ice::Byte*, const Ice::Byte*>&,
                                              const CT&),
                                void (T::*excb)(const Ice::Exception&,
const CT&),
                                void (T::*sentcb)(bool, const CT&) = 0);

template<class T> Callback_Object_ice_invokePtr
newCallback_Object_ice_invoke(const IceUtil::Handle<T>& instance,
                                void (T::*excb)(const Ice::Exception&),
                                void (T::*sentcb)(bool) = 0);

template<class T, typename CT> Callback_Object_ice_invokePtr
newCallback_Object_ice_invoke(const IceUtil::Handle<T>& instance,
                                void (T::*excb)(const Ice::Exception&,
```

```
const CT&),
                                void (T::*sentcb)(bool, const CT&) = 0);
```

The AsyncResult API is deprecated and provided only for backward compatibility. New applications should use the Task API.

The basic mapping is shown below:

```
Ice.AsyncResult<Ice.Callback_Object_ice_invoke>
begin_ice_invoke(
    string operation,
    Ice.OperationMode mode,
    byte[] inParams);

Ice.AsyncResult<Callback_Object_ice_invoke>
begin_ice_invoke(
    string operation,
    Ice.OperationMode mode,
    byte[] inParams,
    Dictionary<string, string> context);

bool end_ice_invoke(out byte[] outParams, AsyncResult r);
```

User exceptions are handled differently than for static asynchronous invocations. Calling `end_ice_invoke` can raise Ice run-time exceptions but never raises user exceptions. Instead, the boolean return value of `end_ice_invoke` indicates whether the operation completed successfully (true) or raised a user exception (false). If the return value is true, the byte sequence contains an encapsulation of the results; otherwise, the byte sequence contains an encapsulation of the user exception.

The generic callback API is also available:

```
Ice.AsyncResult begin_ice_invoke(
    string operation,
    Ice.OperationMode mode,
    byte[] inParams,
    Ice.AsyncCallback cb,
    object cookie);

Ice.AsyncResult begin_ice_invoke(
    string operation,
    Ice.OperationMode mode,
    byte[] inParams,
    Dictionary<string, string> context,
    Ice.AsyncCallback cb,
    object cookie);
```

Refer to the static AMI mapping for a callback example.

For the type-safe callback API, you register callbacks on the `AsyncResult` object just as in the static AMI mapping:

```
public class MyCallback
{
    public void responseCB(bool ret, byte[] results)
    {
        if(ret)
        {
            System.Console.Out.WriteLine("Success");
        }
        else
        {
            System.Console.Out.WriteLine("User exception");
        }
    }

    public void failureCB(Ice.Exception ex)
    {
        System.Console.Err.WriteLine("Exception is: " + ex);
    }
}

...

Ice.AsyncResult<Ice.Callback_Object_ice_invoke> r =
proxy.begin_ice_invoke(...);
MyCallback cb = new MyCallback();
r.whenCompleted(cb.responseCB, cb.failureCB);
```

The caller invokes `whenCompleted` on the `AsyncResult` object and supplies delegates to handle response and failure. The response delegate must match the signature of `Ice.Callback_Object_ice_invoke`:

```
public delegate void Callback_Object_ice_invoke(bool ret, byte[]
outParams);
```

```
System.Threading.Tasks.Task<Ice.Object_Ice_invokeResult>
ice_invokeAsync(string operation,
                Ice.OperationMode mode,
                byte[] inParams,
                Ice.OptionalContext context = new Ice.OptionalContext(),
                System.IProgress<bool> progress = null,
                System.Threading.CancellationToken cancel = new
System.Threading.CancellationToken());
```

The method sends (or queues) an invocation of the given operation and does not block the calling thread. It returns a `Task` that you can use in a number of ways, including blocking to obtain the result, configuring a continuation to be executed when the result becomes available,

and polling to check the status of the request. Refer to the static AMI mapping for more information on the `context`, `progress` and `cancel` arguments.

The `ice_invokeAsync` signature is consistent with the AMI mapping of operations that return multiple values, therefore it uses a structure as its result type:

```
namespace Ice
{
    public struct Object_Ice_invokeResult
    {
        public Object_Ice_invokeResult(bool returnValue, byte[]
outEncaps);
        public bool returnValue;
        public byte[] outEncaps;
    }
}
```

User exceptions are handled differently than for static asynchronous invocations. Calling `ice_invokeAsync` can raise Ice run-time exceptions but never raises user exceptions. Instead, the `returnValue` member of `Object_Ice_invokeResult` indicates whether the operation completed successfully (true) or raised a user exception (false). If `returnValue` is true, the byte sequence in `outEncaps` contains an encapsulation of the results; otherwise, the byte sequence in `outEncaps` contains an encapsulation of the user exception.

```
package com.zeroc.Ice;

public interface Object
{
    public class Ice_invokeResult
    {
        public Ice_invokeResult()
        {
        }

        public Ice_invokeResult(boolean returnValue, byte[] outParams)
        {
            this.returnValue = returnValue;
            this.outParams = outParams;
        }

        public boolean returnValue;
        public byte[] outParams;
    }

    ...
}

public interface ObjectPrx
{

java.util.concurrent.CompletableFuture<com.zeroc.Ice.Object.Ice_invokeR
esult> ice_invokeAsync(
        String operation,
        OperationMode mode,
        byte[] inParams);


java.util.concurrent.CompletableFuture<com.zeroc.Ice.Object.Ice_invokeR
esult> ice_invokeAsync(
        String operation,
        OperationMode mode,
        byte[] inParams,
        java.util.Map<String, String> context);

    ...
}
```

As for statically-typed asynchronous invocations, the return value is a `CompletableFuture`. Its result is an instance of `Object.Ice_invokeResult`. Run-time exceptions cause the future to fail exceptionally, however user exceptions cause the future to succeed: the `returnValue` member of the `Ice_invokeResult` object will be set to `false`, and the `outParams` member contains the encapsulated user exception data.

```
Ice.AsyncResult begin_ice_invoke(
    String operation,
    Ice.OperationMode mode,
    byte[] inParams);

Ice.AsyncResult begin_ice_invoke(
    String operation,
    Ice.OperationMode mode,
    byte[] inParams,
    java.util.Map<String, String> __context);

boolean end_ice_invoke(Ice.ByteSeqHolder outParams, Ice.AsyncResult
__result);
```

User exceptions are handled differently than for static asynchronous invocations. Calling `end_ice_invoke` can raise run-time exceptions but never raises user exceptions. Instead, the boolean return value of `end_ice_invoke` indicates whether the operation completed successfully (true) or raised a user exception (false). If the return value is true, the byte sequence contains an encapsulation of the results; otherwise, the byte sequence contains an encapsulation of the user exception.

The generic callback API is also available:

```
Ice.AsyncResult begin_ice_invoke(
    String operation,
    Ice.OperationMode mode,
    byte[] inParams,
    Ice.Callback cb);

Ice.AsyncResult begin_ice_invoke(
    String operation,
    Ice.OperationMode mode,
    byte[] inParams,
    java.util.Map<String, String> context,
    Ice.Callback cb);
```

Refer to the static AMI mapping for an example of subclassing `Ice.Callback`.

The type-safe callback API looks as follows:

```
Ice.AsyncResult begin_ice_invoke(
    String operation,
    Ice.OperationMode mode,
    byte[] inParams,
    Ice.Callback_Object_ice_invoke cb);

Ice.AsyncResult begin_ice_invoke(
    String operation,
    Ice.OperationMode mode,
    byte[] inParams,
    java.util.Map<String, String> context,
    Ice.Callback_Object_ice_invoke cb);
```

Callers must supply a subclass of `Ice.Callback_Object_ice_invoke`:

```
package Ice;

public abstract class Callback_Object_ice_invoke extends ...
{
    public abstract void response(boolean ret, byte[] outParams);

    public abstract void exception(LocalException ex);
}
```

The boolean argument to `response` indicates whether the operation completed successfully (true) or raised a user exception (false). If the return value is true, the byte sequence contains an encapsulation of the results; otherwise, the byte sequence contains an encapsulation of the user exception.

```
-(id<ICEAsyncResult>) begin_ice_invoke:(NSString*)operation
mode:(ICEOperationMode)mode inEncaps:(NSData*)inEncaps;
-(id<ICEAsyncResult>) begin_ice_invoke:(NSString*)operation
mode:(ICEOperationMode)mode inEncaps:(NSData*)inEncaps
                                    response:(void(^)(BOOL,
NSMutableData*))response exception:(void(^)(ICEException*))exception;
-(id<ICEAsyncResult>) begin_ice_invoke:(NSString*)operation
mode:(ICEOperationMode)mode inEncaps:(NSData*)inEncaps
                                    response:(void(^)(BOOL,
NSMutableData*))response exception:(void(^)(ICEException*))exception
                                    sent:(void(^)(BOOL))sent;
-(BOOL) end_ice_invoke:(NSMutableData**)outEncaps
result:(id<ICEAsyncResult>)result;
```

The `begin_ice_invoke` methods send (or queue) an invocation of the given operation and do not block the calling thread. Refer to the sta tic AMI mapping for more information on the `response`, `exception` and `sent` arguments.

```
def ice_invokeAsync(self, operation, mode, inParams, context=None)
```

The method sends (or queues) an invocation of the given operation and does not block the calling thread. It returns a `Future` that you can use in a number of ways, including blocking to obtain the result, configuring a callback to be executed when the result becomes available, and polling to check the status of the request. Refer to the static AMI mapping for more information on using futures.

Upon completion, the result of the future is a tuple consisting of `(ok, outParams)`, where `ok` is `True` if the invocation completed successfully and `outParams` contains the encapsulated output parameters, or `False` if the invocation resulted in a user exception and `outParams` contains the encapsulated exception.

Using Streams with `ice_invoke`

The streaming interfaces provide the tools an application needs to dynamically invoke operations with arguments of any Slice type. Consider the following Slice definition:

Slice

```
module Calc
{
    exception Overflow
    {
        int x;
        int y;
    }

    interface Compute
    {
        idempotent int add(int x, int y) throws Overflow;
    }
}
```

Now let's write a client that dynamically invokes the `add` operation:

C++11 C++98 C# Java Java Compat

```
std::shared_ptr<Ice::ObjectPrx> proxy = ...
try
{
    std::vector<Ice::Byte> inParams, outParams;

    Ice::OutputStream out(communicator);
    out.startEncapsulation();
    int x = 100, y = -1;
    out.write(x);
    out.write(y);
    out.endEncapsulation();
    out.finished(inParams);

    if(proxy->ice_invoke("add", Ice::OperationMode::Idempotent, inParam
s, outParams))
    {
        // Handle success
        InputStream in(communicator, outParams);
        in.startEncapsulation();
        int result;
        in.read(result);
        in.endEncapsulation();
        assert(result == 99);
    }
    else
    {
        // Handle user exception
    }
}
catch(const Ice::LocalException& ex)
{
    // Handle exception
}
```

```
Ice::ObjectPrx proxy = ...try
{
    std::vector<Ice::Byte> inParams, outParams;

    Ice::OutputStream out(communicator);
    out.startEncapsulation();
    int x = 100, y = -1;
    out.write(x);
    out.write(y);
    out.endEncapsulation();
    out.finished(inParams);

    if(proxy->ice_invoke("add", Ice::Idempotent, inParams, outParams))
    {
        // Handle success
        InputStream in(communicator, outParams);
        in.startEncapsulation();
        int result;
        in.read(result);
        in.endEncapsulation();
        assert(result == 99);
    }
    else
    {
        // Handle user exception
    }
}
catch(const Ice::LocalException& ex)
{
    // Handle exception
}
```

```
Ice.ObjectPrx proxy = ...
try
{
    Ice.OutputStream outStream = new Ice.OutputStream(communicator);
    outStream.startEncapsulation();
    int x = 100, y = -1;
    outStream.writeInt(x);
    outStream.writeInt(y);
    outStream.endEncapsulation();
    byte[] inParams = outStream.finished();

    byte[] outParams;
    if(proxy.ice_invoke("add", Ice.OperationMode.Idempotent, inParams,
out outParams))
    {
        // Handle success
        Ice.InputStream inStream = new Ice.InputStream(communicator,
outParams);
        inStream.startEncapsulation();
        int result = inStream.readInt();
        inStream.endEncapsulation();
        System.Diagnostics.Debug.Assert(result == 99);
    }
    else
    {
        // Handle user exception
    }
}
catch (Ice.LocalException ex)
{
    // Handle exception
}
```

```
com.zeroc.Ice.ObjectPrx proxy = ...
try
{
    com.zeroc.Ice.OutputStream out = new
com.zeroc.Ice.OutputStream(communicator);
    out.startEncapsulation();
    int x = 100, y = -1;
    out.writeInt(x);
    out.writeInt(y);
    out.endEncapsulation();
    byte[] inParams = out.finished();

    com.zeroc.Ice.Object.Ice_invokeResult r = proxy.ice_invoke("add",
com.zeroc.Ice.OperationMode.Idempotent, inParams);
    if(r.returnValue)
    {
        // Handle success
        com.zeroc.Ice.InputStream in = new
com.zeroc.Ice.InputStream(communicator, r.outParams);
        in.startEncapsulation();
        int result = in.readInt();
        in.endEncapsulation();
        assert(result == 99);
    }
    else
    {
        // Handle user exception
    }
}
catch(com.zeroc.Ice.LocalException ex)
{
    // Handle exception
}
```

```
Ice.ObjectPrx proxy = ...
try
{
    Ice.OutputStream out = new Ice.OutputStream(communicator);
    out.startEncapsulation();
    int x = 100, y = -1;
    out.writeInt(x);
    out.writeInt(y);
    out.endEncapsulation();
    byte[] inParams = out.finished();

    Ice.ByteSeqHolder outParams = new Ice.ByteSeqHolder();
    if(proxy.ice_invoke("add", Ice.OperationMode.Idempotent, inParams,
outParams))
    {
        // Handle success
        Ice.InputStream in = new Ice.InputStream(communicator,
outParams.value);
        in.startEncapsulation();
        int result = in.readInt();
        in.endEncapsulation();
        assert(result == 99);
    }
    else
    {
        // Handle user exception
    }
}
catch(Ice.LocalException ex)
{
    // Handle exception
}
```

You can see here that the input and output parameters are enclosed in encapsulations.

We neglected to handle the case of a user exception in this example, so let's implement that now. We assume that we have compiled our program with the Slice-generated code, therefore we can call `throwException` on the input stream and catch `Overflow` directly:

C++11C++98C#JavaJava Compat

```
if(proxy->ice_invoke("add", Ice::OperationMode::Idempotent, inParams,
outParams))
{
    // Handle success
    // ...
}
else
{
    // Handle user exception
    Ice::InputStream in(communicator, outParams);
    try
    {
        in.startEncapsulation();
        in.throwException();
    }
    catch(const Calc::Overflow& ex)
    {
        cout << "overflow while adding " << ex.x
<< " and " << ex.y << endl;
    }
    catch(const Ice::UserException& ex)
    {
        // Handle unexpected user exception
    }
}
```

```
if(proxy->ice_invoke("add", Ice::Idempotent, inParams, outParams))
{
    // Handle success
    // ...
}
else
{
    // Handle user exception
    Ice::InputStream in(communicator, outParams);
    try
    {
        in.startEncapsulation();
        in.throwException();
    }
    catch(const Calc::Overflow& ex)
    {
        cout << "overflow while adding " << ex.x
<< " and " << ex.y << endl;
    }
    catch(const Ice::UserException& ex)
    {
        // Handle unexpected user exception
    }
}
```

```
if(proxy.ice_invoke("add", Ice.OperationMode.Idempotent, inParams, out
outParams))
    {
        // Handle success
        ...
    }
    else
    {
        // Handle user exception
        Ice.InputStream inStream = new Ice.InputStream(communicator,
outParams);
        try
        {
            inStream.startEncapsulation();
            inStream.throwException();
        }
        catch(Calc.Overflow ex)
        {
            System.Console.WriteLine("overflow while adding " +
                                     ex.x + " and " + ex.y);
        }
        catch(Ice.UserException)
        {
            // Handle unexpected user exception
        }
    }
```

```
if(r.returnValue)
    {
        // Handle success
        // ...
    }
    else
    {
        // Handle user exception
        com.zeroc.Ice.InputStream in = new
com.zeroc.Ice.InputStream(communicator, r.outParams);
        try
        {
            in.startEncapsulation();
            in.throwException();
        }
        catch(Calc.Overflow ex)
        {
            System.out.println("overflow while adding " + ex.x + " and "
+ ex.y);
        }
        catch(com.zeroc.Ice.UserException ex)
        {
            // Handle unexpected user exception
        }
    }
```

```
if(proxy.ice_invoke("add", Ice.OperationMode.Idempotent, inParams,
outParams))
    {
        // Handle success
        // ...
    }
    else
    {
        // Handle user exception
        Ice.InputStream in = new Ice.InputStream(communicator,
outParams.value);
        try
        {
            in.startEncapsulation();
            in.throwException();
        }
        catch(Calc.Overflow ex)
        {
            System.out.println("overflow while adding " + ex.x + " and "
+ ex.y);
        }
        catch(Ice.UserException ex)
        {
            // Handle unexpected user exception
        }
    }
```

This is obviously a contrived example: if the Slice-generated code is available, why bother using dynamic dispatch? In the absence of Slice-generated code, the caller would need to manually unmarshal the user exception, which is outside the scope of this manual.

As a defensive measure, the code traps `UserException`. This could be raised if the Slice definition of `add` is modified to include another user exception but this segment of code did not get updated accordingly.

See Also

- Request Contexts
- Streaming Interfaces

Dynamic Dispatch

This page describes the server-side language mappings for the `ice_invoke` operation on Blobject servant classes.

On this page:

Synchronous Mapping for `ice_invoke`

Implementing the dynamic dispatch model requires writing a subclass of an Ice class and defining the `ice_invoke` function. We continue using the `Compute` interface from our dynamic invocation example to demonstrate the server-side implementation.

`ice_invoke` API

The synchronous mapping for `ice_invoke` is shown below:

C++11 C++11 Array C++98 C++98 Array C# Java Java Compat ObjC Python

```
class ComputeI : public Ice::Blobject
{
public:

    virtual bool ice_invoke(std::vector<Ice::Byte> inParams,
                            std::vector<Ice::Byte>& outParams,
                            const Ice::Current& current) override;
};
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

```
class ComputeI : public Ice::BlobjectArray
{
public:
    virtual bool ice_invoke(std::pair<const Ice::Byte*, const
Ice::Byte*> in,
                            std::vector<Ice::Byte>& out,
                            const Ice::Current& current) override;
};
```

The `BlobjectArray` class uses an alternative mapping for `sequence` input parameters that avoids the overhead of extra copying. The `ice_invoke` function treats the encoded input parameters as a value of type `sequence<byte>`.

An instance of `ComputeI` is an Ice object because `BlobjectArray` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

```
class ComputeI : public Ice::Blobject
{
public:

    virtual bool ice_invoke(const std::vector<Ice::Byte>& inParams,
                            std::vector<Ice::Byte>& outParams,
                            const Ice::Current& current);
};
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

```
class ComputeI : public Ice::BlobjectArray
{
public:
    virtual bool ice_invoke(const std::pair<const Ice::Byte*, const
Ice::Byte*>& in,
                            std::vector<Ice::Byte>& out,
                            const Ice::Current& current) = 0;
};
```

The `BlobjectArray` class uses an alternative mapping for `sequence` input parameters that avoids the overhead of extra copying. The `ice_invoke` function treats the encoded input parameters as a value of type `sequence<byte>`.

An instance of `ComputeI` is an Ice object because `BlobjectArray` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

```
public class ComputeI : Ice.Blobject
{
    public bool ice_invoke(byte[] inParams, out byte[] outParams,
Ice.Current current);
    {
        // ...
    }
}
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

```
public class ComputeI implements com.zeroc.Ice.Blobject
{
    public com.zeroc.Ice.Object.Ice_invokeResult ice_invoke(byte[]
inParams, com.zeroc.Ice.Current current)
        throws com.zeroc.Ice.UserException
    {
        // ...
    }
}
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

```
public class ComputeI extends Ice.Blobject
{
    public boolean ice_invoke(
        byte[] inParams,
        Ice.ByteSeqHolder outParams,
        Ice.Current current)
        throws Ice.UserException
    {
        // ...
    }
}
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

```
@interface ComputeI : ICEBlobject<ICEBlobject>
-(BOOL) ice_invoke:(NSData*)inEncaps
outEncaps:(NSMutableData**)outEncaps current:(ICECurrent*)current;
@end
```

An instance of `ComputeI` is an Ice object because `ICEBlobject` derives from `ICEObject`, therefore an instance can be added to an object adapter like any other servant.

```
class ComputeI(Ice.Blobject):
    def ice_invoke(self, inParams, current):
        # ...
```

An instance of `ComputeI` is an Ice object because `Ice.Blobject` derives from `Ice.Object`, therefore an instance can be added to an object adapter like any other servant.

Object Operations

For the purposes of this discussion, the implementation of `ice_invoke` handles only the `add` operation and raises `OperationNotExistException` for all other operations. In a real implementation, the servant must also be prepared to receive invocations of the following `Object` operations:

- `string ice_id()`
  Returns the Slice type ID of the servant's most-derived type.
- `StringSeq ice_ids()`
  Returns a sequence of strings representing all of the Slice interfaces supported by the servant, including `"::Ice::Object"`.
- `bool ice_isA(string id)`
  Returns `true` if the servant supports the interface denoted by the given Slice type ID, or `false` otherwise. This operation is invoked by the proxy function `checkedCast`.
- `void ice_ping()`
  Verifies that the object denoted by the identity and facet contained in `Ice::Current` is reachable.

Implementing `ice_invoke`

Here is our simplified implementation of `ice_invoke`:

C++11 C++98 C# Java Java Compat

```
bool ComputeI::ice_invoke(std::vector<Ice::Byte> inParams,
                          std::vector<Ice::Byte>& outParams,
                          const Ice::Current& current)
{
    if(current.operation == "add")
    {
        auto communicator = current.adapter->getCommunicator();
        Ice::InputStream in(communicator, inParams);
        in.startEncapsulation();
        int x, y;
        in.read(x);
        in.read(y);
        in.endEncapsulation();

        Ice::OutputStream out(communicator);
        if(checkOverflow(x, y))
        {
            Calc::Overflow ex(x, y);
            out.startEncapsulation();
            out.writeException(ex);
            out.endEncapsulation();
            out.finished(outParams);
            return false;
        }
        else
        {
            out.startEncapsulation();
            out.write(x + y);
            out.endEncapsulation();
            out.finished(outParams);
            return true;
        }
    }
    else
    {
        throw Ice::OperationNotExistException(__FILE__, __LINE__,
current.id, current.facet, current.operation);
    }
}
```

```
bool ComputeI::ice_invoke(const std::vector<Ice::Byte>& inParams,
                          std::vector<Ice::Byte>& outParams,
                          const Ice::Current& current)
{
    if(current.operation == "add")
    {
        Ice::CommunicatorPtr communicator =
current.adapter->getCommunicator();
        Ice::InputStream in(communicator, inParams);
        in.startEncapsulation();
        Ice::Int x, y;
        in.read(x);
        in.read(y);
        in.endEncapsulation();

        Ice::OutputStream out(communicator);
        if(checkOverflow(x, y))
        {
            Calc::Overflow ex(x, y);
            out.startEncapsulation();
            out.writeException(ex);
            out.endEncapsulation();
            out.finished(outParams);
            return false;
        }
        else
        {
            out.startEncapsulation();
            out.write(x + y);
            out.endEncapsulation();
            out.finished(outParams);
            return true;
        }
    }
    else
    {
        throw Ice::OperationNotExistException(__FILE__, __LINE__,
current.id, current.facet, current.operation);
    }
}
```

```
public bool ice_invoke(byte[] inParams, out byte[] outParams,
Ice.Current current);
    {
        if(current.operation.Equals("add"))
        {
            Ice.Communicator communicator =
```

```
current.adapter.getCommunicator();
            Ice.InputStream inStream = new Ice.InputStream(communicator,
inParams);
            inStream.startEncapsulation();
            int x = inStream.readInt();
            int y = inStream.readInt();
            inStream.endEncapsulation();

            Ice.OutputStream outStream = new
Ice.OutputStream(communicator);
            try
            {
                if(checkOverflow(x, y))
                {
                    Calc.Overflow ex = new Calc.Overflow();
                    ex.x = x;
                    ex.y = y;
                    outStream.startEncapsulation();
                    outStream.writeException(ex);
                    outStream.endEncapsulation();
                    outParams = outStream.finished();
                    return false;
                }
                else
                {
                    outStream.startEncapsulation();
                    outStream.writeInt(x + y);
                    outStream.endEncapsulation();
                    outParams = outStream.finished();
                    return true;
                }
            }
            finally
            {
                outStream.destroy();
            }
        }
        else
        {
            Ice.OperationNotExistException ex = new
Ice.OperationNotExistException();
            ex.id = current.id;
            ex.facet = current.facet;
            ex.operation = current.operation;
```

```
            throw ex;
        }
    }
```

```
public com.zeroc.Ice.Object.Ice_invokeResult ice_invoke(byte[] inParams,
com.zeroc.Ice.Current current)
    {
        if(current.operation.equals("add"))
        {
            com.zeroc.Ice.Communicator communicator =
current.adapter.getCommunicator();
            com.zeroc.Ice.InputStream in = new
com.zeroc.Ice.InputStream(communicator, inParams);
            in.startEncapsulation();
            int x = in.readInt();
            int y = in.readInt();
            in.endEncapsulation();

            com.zeroc.Ice.OutputStream out = new
com.zeroc.Ice.OutputStream(communicator);
            com.zeroc.Ice.Object.Ice_invokeResult r = new
com.zeroc.Ice.Object.Ice_invokeResult();
            if(checkOverflow(x, y))
            {
                Calc.Overflow ex = new Calc.Overflow();
                ex.x = x;
                ex.y = y;
                out.startEncapsulation();
                out.writeException(ex);
                out.endEncapsulation();
                r.outParams = out.finished();
                r.returnValue = false;
            }
            else
            {
                out.startEncapsulation();
                out.writeInt(x + y);
                out.endEncapsulation();
                r.outParams = out.finished();
                r.returnValue = true;
            }
            return r;
        }
        else
        {
            Ice.OperationNotExistException ex = new
Ice.OperationNotExistException();
            ex.id = current.id;
            ex.facet = current.facet;
            ex.operation = current.operation;
            throw ex;
        }
    }
```

```
public boolean ice_invoke(
        byte[] inParams,
        Ice.ByteSeqHolder outParams,
        Ice.Current current)
    {
        if(current.operation.equals("add"))
        {
            Ice.Communicator communicator =
current.adapter.getCommunicator();
            Ice.InputStream in = new Ice.InputStream(communicator,
inParams);
            in.startEncapsulation();
            int x = in.readInt();
            int y = in.readInt();
            in.endEncapsulation();

            Ice.OutputStream out = new Ice.OutputStream(communicator);
            if(checkOverflow(x, y))
            {
                Calc.Overflow ex = new Calc.Overflow();
                ex.x = x;
                ex.y = y;
                out.startEncapsulation();
                out.writeException(ex);
                out.endEncapsulation();
                outParams.value = out.finished();
                return false;
            }
            else
            {
                out.startEncapsulation();
                out.writeInt(x + y);
                out.endEncapsulation();
                outParams.value = out.finished();
                return true;
            }
        }
        else
        {
            Ice.OperationNotExistException ex = new
Ice.OperationNotExistException();
            ex.id = current.id;
            ex.facet = current.facet;
            ex.operation = current.operation;
            throw ex;
        }
    }
```

If an overflow is detected, the code "raises" the `Calc::Overflow` user exception by calling `writeException` on the output stream and returning `false`, otherwise the return value is encoded and the function returns `true`.

Asynchronous Mapping for `ice_invoke`

Ice provides an alternate base class for asynchronous dispatch:

C++11 C++11 Array C++98 C++98 Array C# Java Java Compat Python

```
namespace Ice
{
    class BlobjectAsync : public virtual Ice::Object
    {
    public:

        virtual void ice_invokeAsync(std::vector<Byte> inParams,
                                     std::function<void(bool, const
std::vector<Byte>&)> response,

std::function<void(std::exception_ptr)> eptr,
                                     const Ice::Current& current) = 0;
    }
}
```

You need to create a servant class derived from `BlobjectAsync`, and override `ice_invokeAsync`.

Upon a successful invocation, the servant must call `response`, passing `true` as the first argument and the encapsulated operation result into the second argument. The servant calls `response` with `false` as the first argument to report a user exception; the second argument is then the encapsulated user exception.

The `eptr` function is used to report Ice run-time exceptions and other (non-Ice) C++ exceptions. If you accidentally report a user exception with this function, the caller will receive a `UnknownUserException`.

```
namespace Ice
{
    class BlobjectArrayAsync : public virtual Object
    {
    public:

        virtual void ice_invokeAsync(std::pair<const Ice::Byte*, const
Ice::Byte*> inParams,
                                     std::function<void(bool, const
std::pair<const Ice::Byte*, const Ice::Byte*>&)> response,

std::function<void(std::exception_ptr)> eptr,
                                     const Ice::Current& current) = 0;
    };
}
```

The `BlobjectArrayAsync` class uses an alternative mapping for `sequence` input parameters that avoids the overhead of extra copying. The `ice_invokeAsync` function treats the encoded input parameters as a value of type `sequence<byte>`.

You need to create a servant class derived from `BlobjectArrayAsync`, and override `ice_invokeAsync.`

Upon a successful invocation, the servant must call `response`, passing `true` as the first argument and the encapsulated operation result into the second argument. The servant calls `response` with `false` as the first argument to report a user exception; the second argument is then the encapsulated user exception.

The `eptr` function is used to report Ice run-time exceptions and other (non-Ice) C++ exceptions. If you accidentally report a user exception with this function, the caller will receive a `UnknownUserException`.

```
namespace Ice
{
    class BlobjectAsync : public virtual Ice::Object
    {
    public:


virtual void ice_invoke_async(const AMD_Object_ice_invokePtr& cb,

 const std::vector<Ice::Byte>& inParams,
                                       const Ice::Current& current) = 0;
    }
}
```

You need to create a class derived from `BlobjectAsync`, and override `ice_invoke_async.` The first argument to the servant's member function is a callback object of type `Ice::AMD_Object_ice_invoke`, shown here:

```
namespace Ice
{
    class AMD_Object_ice_invoke : ...
    {
    public:
        virtual void ice_response(bool result,
const std::vector<Ice::Byte>& outParams) = 0;
        virtual void ice_response(bool result,
const std::pair<const Ice::Byte*, const Ice::Byte*>& outParams) = 0;

        virtual void ice_exception(const std::exception&) = 0;
        virtual void ice_exception() = 0;
    }
}
```

Upon a successful invocation, the servant must call `ice_response`, passing `true` as the first argument and the encapsulated operation result into the second argument. The servant calls `ice_response` with `false` as the first argument to report a user exception; the second argument is then the encapsulated user exception.

`ice_exception` is used to report Ice run-time exceptions and other (non-Ice) standard C++ exceptions. If you accidentally report a user exception with `ice_exception`, the caller will receive a `UnknownUserException`.

```
namespace Ice
{
    class BlobjectArrayAsync : public virtual Object
    {
    public:
        virtual void ice_invoke_async(const
Ice:AMD_Object_ice_invokePtr& cb,
                                      const std::pair<const Ice::Byte*,
const Ice::Byte*>& inParams,
                                      const Ice::Current& current) = 0;
    };
}
```

The `BlobjectArrayAsync` class uses an alternative mapping for `sequence` input parameters that avoids the overhead of extra copying. The `ice_invoke_async` function treats the encoded input parameters as a value of type `sequence<byte>`.

You need to create a class derived from `BlobjectArrayAsync`, and override `ice_invoke_async.` The first argument to the servant's member function is a callback object of type `Ice::AMD_Object_ice_invoke`, shown here:

```
namespace Ice
{
    class AMD_Object_ice_invoke : ...
    {
    public:
        virtual void ice_response(bool result,
const std::vector<Ice::Byte>& outParams) = 0;
        virtual void ice_response(bool result,
const std::pair<const Ice::Byte*, const Ice::Byte*>& outParams) = 0;

        virtual void ice_exception(const std::exception&) = 0;
        virtual void ice_exception() = 0;
    }
}
```

Upon a successful invocation, the servant must call `ice_response`, passing `true` as the first argument and the encapsulated operation result into the second argument. The servant calls `ice_response` with `false` as the first argument to report a user exception; the second argument is then the encapsulated user exception.

`ice_exception` is used to report Ice run-time exceptions and other (non-Ice) standard C++ exceptions. If you accidentally report a user exception with `ice_exception`, the caller will receive a `UnknownUserException`.

```
namespace Ice
{
    public struct Object_Ice_invokeResult
    {
        public Object_Ice_invokeResult(bool returnValue, byte[]
outEncaps);
        public bool returnValue;
        public byte[] outEncaps;
    }

    public abstract class BlobjectAsync : Ice.ObjectImpl
    {
        public abstract Task<Object_Ice_invokeResult>
ice_invokeAsync(byte[] inEncaps, Current current);
    }
}
```

To implement asynchronous dynamic dispatch, a server must subclass `BlobjectAsync` and override `ice_invokeAsync`.

The return value for successful completion, or for a user exception, is a `Task` whose result is an instance of `Object_Ice_invokeResult`.

The servant may optionally raise a user exception directly and the Ice run time will marshal it for you.

```
package com.zeroc.Ice;

public interface BlobjectAsync extends com.zeroc.Ice.Object
{
    java.util.concurrent.CompletionStage<Object.Ice_invokeResult>
ice_invokeAsync(byte[] inEncaps, Current current)
        throws UserException;
}
```

To implement asynchronous dynamic dispatch, a server must implement `BlobjectAsync` and define `ice_invokeAsync`.

The return value for successful completion, or for a user exception, is a `CompletionStage` whose result is an instance of `Object.Ice_invokeResult`:

```
package com.zeroc.Ice;

public interface Object
{
    public class Ice_invokeResult
    {
        public Ice_invokeResult()
        {
        }

        public Ice_invokeResult(boolean returnValue, byte[] outParams)
        {
            this.returnValue = returnValue;
            this.outParams = outParams;
        }

        public boolean returnValue;
        public byte[] outParams;
    }

    ...
}
```

The servant may optionally raise a user exception directly and the Ice run time will marshal it for you.

```
package Ice;

public abstract class BlobjectAsync extends Ice.ObjectImpl
{
    public abstract void ice_invoke_async(
        Ice.AMD_Object_ice_invoke cb,
        byte[] inParams,
        Ice.Current current);

    // ...
}
```

To implement asynchronous dynamic dispatch, a server must subclass `BlobjectAsync` and override `ice_invoke_async`.

As with any other asynchronous operation, the first argument to the servant's member function is always a callback object. In this case, the callback object is of type `Ice.AMD_Object_ice_invoke`, shown here:

```
package Ice;

public interface AMD_Object_ice_invoke
{
    void ice_response(boolean result, byte[] outParams);
    void ice_exception(java.lang.Exception ex);
}
```

Upon a successful invocation, the servant must invoke `ice_response` on the callback object, passing `true` as the first argument and encoding the encapsulated operation results into `outParams`. To report a user exception, the servant invokes `ice_response` with `false` a s the first argument and the encapsulated form of the exception in `outParams`. Alternatively, the servant can pass a user exception instance to `ice_exception`.

```
class Blobject(Object):
    def ice_invoke(self, inParams, current):
        # ...
```

To implement asynchronous dynamic dispatch, a server must subclass `Blobject` and implement `ice_invoke`.

The return value for successful completion, or for a user exception, is a `Future` whose result is a tuple consisting of `(ok, outParams)`, where `ok` is `True` if the invocation completed successfully and `outParams` contains the encapsulated output parameters, or `False` if the invocation resulted in a user exception and `outParams` contains the encapsulated exception.

The servant may optionally raise a user exception directly and the Ice run time will marshal it for you.

See Also

- Object Adapters
- Type IDs
- Object Identity
- Versioning

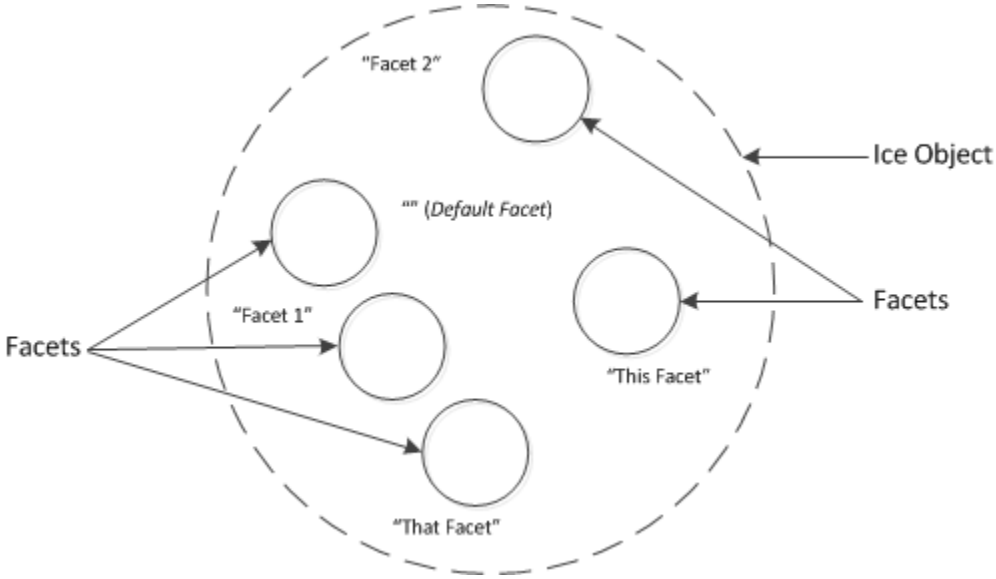
Facets

Facets provide a general-purpose mechanism for non-intrusively extending the type system of an application, by loosely coupling new type instances to existing ones. This shifts the type selection process from compile to run time and implements a form of late binding. This is particularly useful for versioning an application.

On this page:

Ice Objects as Collections of Facets

Up to this point, we have presented an Ice object as a single conceptual entity, that is, as an object with a single most-derived interface and a single identity, with the object being implemented by a single servant. However, an Ice object is more correctly viewed as a collection of one or more sub-objects known as facets, as shown below:



An Ice object with five facets sharing a single object identity.

The diagram above shows a single Ice object with five facets. Each facet has a name, known as the *facet name*. Within a single Ice object, all facets must have unique names. Facet names are arbitrary strings that are assigned by the server that implements an Ice object. A facet with an empty facet name is legal and known as the *default facet*. Unless you arrange otherwise, an Ice object has a single default facet; by default, operations that involve Ice objects and servants operate on the default facet.

Note that all the facets of an Ice object share the same single identity, but have different facet names. Recall the definition of `Ice::Current` once more:

Slice

```
module Ice
{
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Nonmutating, \Idempotent }

    local struct Current
    {
        ObjectAdapter   adapter;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
        Context         ctx;
        int             requestId;
        EncodingVersion encoding;
    }
}
```

By definition, if two facets have the same `id` field, they are part of the same Ice object. Also by definition, if two facets have the same `id` fiel d, their `facet` fields have different values.

Even though Ice objects usually consist of just the default facet, it is entirely legal for an Ice object to consist of facets that all have non-empty names (that is, it is legal for an Ice object not to have a default facet).

Each facet has a single most-derived interface. There is no need for the interface types of the facets of an Ice object to be unique. It is legal for two facets of an Ice object to implement the same most-derived interface.

Each facet is implemented by a servant. All the usual implementation techniques for servants are available to implement facets — for example, you can implement a facet using a servant locator. Typically, each facet of an Ice object has a separate servant, although, if two facets of an Ice object have the same type, they can also be implemented by a single servant (for example, using a default servant).

Server-Side Facet Operations

On the server side, the object adapter offers a number of operations to support facets:

Slice

```
namespace Ice
{
    dictionary<string, Object> FacetMap;

    local interface ObjectAdapter
    {
        Object*  addFacet(Object servant, Identity id, string facet);
        Object*  addFacetWithUUID(Object servant, string facet);
        Object   removeFacet(Identity id, string facet);
        Object   findFacet(Identity id, string facet);

        FacetMap findAllFacets(Identity id);
        FacetMap removeAllFacets(Identity id);
        // ...
    }
}
```

These operations have the same semantics as the corresponding "normal" operations for servant activation and deactivation (`add`, `addWithUUID`, `remove`, and `find`), but also accept a facet name. The corresponding "normal" operations are simply convenience operations that supply an empty facet name. For example, `remove(id)` is equivalent to `removeFacet(id, "")`, that is, `remove(id)` operates on the default facet.

`findAllFacets` returns a dictionary of <*facet-name, servant*> pairs that contains all the facets for the given identity.

`removeAllFacets` removes all facets for a given identity from the active servant map, that is, it removes the corresponding Ice object entirely. The operation returns a dictionary of <*facet-name, servant*> pairs that contains all the removed facets.

These operations are sufficient for the server to create Ice objects with any number of facets. For example, assume that we have the following Slice definitions:

Slice

```
module Filesystem
{
    // ...

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }
}

module FilesystemExtensions
{
    // ...

    class DateTime extends TimeOfDay
    {
            // ...
    }

    struct Times
    {
        DateTime createdDate;
        DateTime accessedDate;
        DateTime modifiedDate;
    }

    interface Stat
    {
        idempotent Times getTimes();
    }
}
```

Here, we have a `File` interface that provides operations to read and write a file, and a `Stat` interface that provides access to the file creation, access, and modification time. (Note that the `Stat` interface is defined in a different module and could also be defined in a different source file.) If the server wants to create an Ice object that contains a `File` instance as the default facet and a `Stat` instance that provides access to the time details of the file, it could do so as follows:

C++11 C++98

```
// Create a File instance.
//
auto file = std::make_shared<FileI>();

// Create a Stat instance.
//
auto dt = std::make_shared<FilesystemExtensions::DateTime>();
FilesystemExtensions::Times times;
times.createdDate = dt;
times.accessedDate = dt;
times.modifiedDate = dt;
auto stat = std::make_shared<StatI>(times);

// Register the File instance as the default facet.
//
auto filePrx = myAdapter->addWithUUID(file);

// Register the Stat instance as a facet with name "Stat".
//
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

```
// Create a File instance.
//
Filesystem::FilePtr file = new FileI;

// Create a Stat instance.
//
FilesystemExctensions::DateTimePtr dt
= new FilesystemExtensions::DateTime;
FilesystemExtensions::Times times;
times.createdDate = dt;
times.accessedDate = dt;
times.modifiedDate = dt;
FilesystemExtensions::StatPtr stat = new StatI(times);

// Register the File instance as the default facet.
//
Ice::ObjectPrx filePrx = myAdapter->addWithUUID(file);

// Register the Stat instance as a facet with name "Stat".
//
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

The first few lines simply create and initialize a `FileI` and `StatI` instance. (The details of this do not matter here.) All the action is in the last two statements:

C++11 C++98

```
auto filePrx = myAdapter->addWithUUID(file);
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

```
Ice::ObjectPrx filePrx = myAdapter->addWithUUID(file);
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

This registers the `FileI` instance with the object adapter as usual. (In this case, we let the Ice run time generate a UUID as the object identity.) Because we are calling `addWithUUID` (as opposed to `addFacetWithUUID`), the instance becomes the default facet.

The second line adds a facet to the instance with the facet name `Stat`. Note that we call `ice_getIdentity` on the `File` proxy to pass an object identity to `addFacet`. This guarantees that the two facets share the same object identity.

Note that, in general, it is a good idea to use `ice_getIdentity` to obtain the identity of an existing facet when adding a new facet. That way, it is guaranteed that the facets share the same identity. (If you accidentally pass a different identity to `addFacet`, you will not add a facet to an existing Ice object, but instead register a new Ice object; using `ice_getIdentity` makes this mistake impossible.)

Client-Side Facet Operations

On the client side, which facet a request is addressed to is implicit in the proxy that is used to send the request. For an application that does not use facets, the facet name is always empty so, by default, requests are sent to the default facet.

The client can use a `checkedCast` to obtain a proxy for a particular facet. For example, assume that the client obtains a proxy to a `File` ins tance as shown above. The client can cast between the `File` facet and the `Stat` facet (and back) as follows:

C++11 C++98

```
// Get a File proxy.
//
std::shared_ptr<Filesystem::FilePrx> file = ...;

// Get the Stat facet.
//
auto stat =
Ice::checkedCast<FilesystemExtensions::StatPrx>(file, "Stat");

// Go back from the Stat facet to the File facet.
//
auto file2 = Ice::checkedCast<Filesystem::FilePrx>(stat, "");

assert(Ice::proxyIdentityAndFacetEqual(file, file2));
```

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Get the Stat facet.
//
FilesystemExtensions::StatPrx stat =
FilesystemExtensions::StatPrx::checkedCast(file, "Stat");

// Go back from the Stat facet to the File facet.
//
Filesystem::FilePrx file2 = Filesystem::FilePrx::checkedCast(stat, "");

assert(file2 == file); // The two proxies are equal in C++98
```

This example illustrates that, given any facet of an Ice object, the client can navigate to any other facet by using a `checkedCast` with the facet name.

If an Ice object does not provide the specified facet, `checkedCast` returns null:

C++11 C++98

```
auto stat =
Ice::checkedCast<FilesystemExtensions::StatPrx>(file, "Stat");

if(!stat)
{
    // No Stat facet on this object, handle error...
}
else
{
    auto times = stat->getTimes();

    // Use times struct...
}
```

```
FilesystemExtensions::StatPrx stat =
FilesystemExtensions::StatPrx::checkedCast(file, "Stat");

if(!stat)
{
    // No Stat facet on this object, handle error...
}
else
{
    FilesystemExtensions::Times times = stat->getTimes();

    // Use times struct...
}
```

Note that `checkedCast` also returns a null proxy if a facet exists, but the cast is to the wrong type. For example:

C++11 C++98

```
// Get a File proxy.
//
std::shared_ptr<Filesystem::FilePrx> file = ...;

// Cast to the wrong type.
//
auto prx = Ice::checkedCast<SomeTypePrx>(file, "Stat");

assert(!prx); // checkedCast returns a null proxy.
```

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Cast to the wrong type.
//
SomeTypePrx prx = SomeTypePrx::checkedCast(file, "Stat");

assert(!prx); // checkedCast returns a null proxy.
```

If you want to distinguish between non-existence of a facet and the facet being of the incorrect type, you can first obtain the facet as type `Object` and then down-cast to the correct type:

C++11 C++98

```
// Get a File proxy.
//
std::shared_ptr<Filesystem::FilePrx> file = ...;

// Get the facet as type Object.
//
auto obj = Ice::checkedCast<ObjectPrx>(file, "Stat");
if(!obj)
{
    // No facet with name "Stat" on this Ice object.
}
else
{
    auto stat = Ice::checkedCast<FilesystemExtensions::StatPrx>(file);
    if(!stat)
    {
        // There is a facet with name "Stat", but it is not
        // of type FilesystemExtensions::Stat.
    }
    else
    {
        // Use stat...
    }
}
```

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Get the facet as type Object.
//
Ice::ObjectPrx obj = Ice::ObjectPrx::checkedCast(file, "Stat");
if(!obj)
{
    // No facet with name "Stat" on this Ice object.
}
else
{
    FilesystemExtensions::StatPrx stat =
FilesystemExtensions::StatPrx::checkedCast(file);
    if(!stat)
    {
        // There is a facet with name "Stat", but it is not
        // of type FilesystemExtensions::Stat.
    }
    else
    {
        // Use stat...
    }
}
```

This last example also illustrates that

C++11 C++98

```
Ice::checkedCast<StatPrx>(prx, "")
```

```
StatPrx::checkedCast(prx, "")
```

is *not* the same as

C++11 C++98

```
Ice::checkedCast<StatPrx>(prx)
```

```
StatPrx::checkedCast(prx)
```

The first version explicitly requests a cast to the default facet. This means that the Ice run time first looks for a facet with the empty name and then attempts to down-cast that facet (if it exists) to the type `Stat`.

The second version requests a down-cast that *preserves* whatever facet is currently effective in the proxy. For example, if the `prx` proxy currently holds the facet name "Joe", then (if `prx` points at an object of type `Stat`) the run time returns a proxy of type `StatPrx` that also stores the facet name "Joe".

It follows that, to navigate between facets, you must always use the two-argument version of `checkedCast`, whereas, to down-cast to another type while preserving the facet name, you must always use the single-argument version of `checkedCast`.

You can always check what the current facet of a proxy is by calling `ice_getFacet`:

C++11 C++98

```
std::shared_ptr<Ice::ObjectPrx> obj = ...;

cout << obj->ice_getFacet() << endl; // Print facet name
```

```
Ice::ObjectPrx obj = ...;

cout << obj->ice_getFacet() << endl; // Print facet name
```

This prints the facet name. (For the default facet, `ice_getFacet` returns the empty string.)

Facet Exception Semantics

The common exceptions `ObjectNotExistException` and `FacetNotExistException` have the following semantics:

- `ObjectNotExistException`
  This exception is raised only if no facets exist at all for a given object identity.
- `FacetNotExistException`
  This exception is raised only if at least one facet exists for a given object identity, but not the specific facet that is the target of an operation invocation.

If you are using servant locators or default servants, you must take care to preserve these semantics. In particular, if you return null from a servant locator's `locate` operation, this appears to the client as an `ObjectNotExistException`. If the object identity for a request is known (that is, there is at least one facet with that identity), but no facet with the specified name exists, you must explicitly throw a `FacetNotExistException` from `locate` instead of simply returning null.

See Also

- Object Identity
- Run-Time Exceptions
- Object Adapters
- Servant Locators
- Default Servants
- The Current Object

Versioning

Once you have developed and deployed a distributed application, and once the application has been in use for some time, it is likely that you will want to make some changes to the application. For example, you may want to add new functionality to a later version of the application, or you may want to change some existing aspect of the application. Of course, ideally, such changes are accomplished without breaking already deployed software, that is, the changes should be backward compatible. Evolving an application in this way is generally known as *ve rsioning*.

Topics

Versioning through Incremental Updates

Ice allows you to update your applications and Slice definitions in a backwards-compatible manner - in such a way that the existing applications (using the existing Slice definitions) and the updated applications (using the updated Slice definitions) can easily and safely communicate with each other.

You can safely update your Slice using the techniques described on this page:

- Adding Operations
- Optional Operation Parameters and Class Data Members

Adding Operations

Suppose that we have deployed our file system application and want to add extra functionality to a new version. Specifically, let us assume that the original version (version 1) only provides the basic functionality to use files, but does not provide extra information, such as the modification date or the file size. The question is then, how can we upgrade the existing application with this new functionality? Here is a small excerpt of the original (version 1) Slice definitions once more:

Slice

```
// Version 1
module Filesystem
{
    // ...

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }
}
```

Your first attempt at upgrading the application might look as follows:

Slice

```
// Version 2
module Filesystem
{
    // ...

    class DateTime extends TimeOfDay    // New in version 2
    {
        // ...
    }

    class Times                         // New in version 2
    {
        DateTime createdDate;
        DateTime accessedDate;
    }

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;

        idempotent Times getTimes();    // New in version 2
    }
}
```

Note that the version 2 definition does not change anything that was present in version 1; instead, it only adds two new types and adds an operation to the `File` interface. Version 1 clients can continue to work with both version 1 and version 2 `File` objects because version 1 clients do not know about the `getTimes` operation and therefore will not call it; version 2 clients, on the other hand, can take advantage of the new functionality. The reason this works is that the Ice protocol invokes an operation by sending the operation name as a string on the wire (rather than using an ordinal number or hash value to identify the operation). Ice guarantees that any future version of the protocol will retain this behavior, so it is safe to add a new operation to an existing interface without recompiling all clients.

However, this approach contains a pitfall: the tacit assumption built into this approach is that no version 2 client will ever use a version 1 object. If the assumption is violated (that is, a version 2 client uses a version 1 object), the version 2 client will receive an `OperationNotExistException` when it invokes the new `getTimes` operation because that operation is supported only by version 2 objects.

Whether you can make this assumption depends on your application. In some cases, it may be possible to ensure that version 2 clients will never access a version 1 object, for example, by simultaneously upgrading all servers from version 1 to version 2, or by taking advantage of application-specific constraints that ensure that version 2 clients only contact version 2 objects. However, for some applications, doing this is impractical.

Note that you could write version 2 clients to catch and react to an `OperationNotExistException` when they invoke the `getTimes` operation: if the operation succeeds, the client is dealing with a version 2 object, and if the operation raises `OperationNotExistsException`, the client is dealing with a version 1 object.

Optional Operation Parameters and Class Data Members

You can safely add one or more optional parameters to an operation without breaking clients or servers that don't know anything of these new parameters.

For example, we could add an optional parameter to the `read` operation in our Filesystem `File` interface to optionally limit the number of lines returned:

Slice

```
interface File extends Node
{
    idempotent Lines read(optional (1) int max);
    ...
}
```

A client using the updated Slice definition can set this maximum value, and if its request is received by a new server, the server will receive this max value and should behave accordingly. An old server wouldn't receive this optional parameter, and would continue to behave as before.

You can likewise add optional data members to an existing class without breaking existing applications that use this class. Let's go back to the FileSystem version 2 presented above - say we need to add a third data member to Times, while preserving existing applications that use the existing two-data-members Times. The solution is to add an optional data member, for example:

Slice

```
 class Times
{
     DateTime createdDate;
     DateTime accessedDate;
     optional (1) DateTime modifiedDate;
 }
```

Older applications that know nothing about `modifiedDate` will continue to receive, create, send (etc.) `Times` instances as before (without a `modifiedDate` data member), while newer applications can set and retrieve the optional `modifiedDate` member.

See Also

- Slice for a Simple File System
- Optional Values

Versioning with Facets

In the most general sense, facets provide a mechanism for implementing multiple interfaces for a single object. The key point is that, to add a new interface to an object, none of the existing definitions have to be touched, so no compatibility issues can arise. More importantly, the decision as to which facet to use is made at run time instead of at compile time.

Used judiciously, facets can handle versioning requirements when the incremental updates approach described earlier is no longer sufficient.

On this page:

Facet Selection

Given that we have decided to extend an application with facets, we have to deal with the question of how clients select the correct facet. The answer typically involves an explicit selection of a facet sometime during client start-up. For example, in our file system application, clients always begin their interactions with the file system by creating a proxy to the root directory. Let us assume that our versioning requirements have led to version 1 and version 2 definitions of directories as follows:

Slice

```
module Filesystem // Original version
{
// ...

    interface Directory extends Node
    {
        idempotent NodeSeq list();
        // ...
    }
}

module FilesystemV2
{
    // ...

    enum NodeType { Directory, File }

    class NodeDetails
    {
        NodeType type;
        string name;
        DateTime createdTime;
        DateTime accessedTime;
        DateTime modifiedTime;
        // ...
    }

    interface Directory extends Filesystem::Node
    {
       idempotent NodeDetailsSeq list();
       // ...
    }
}
```

In this case, the semantics of the `list` operation have changed in version 2. A version 1 client uses the following code to obtain a proxy to the root directory:

C++11 C++98

```
// Create a proxy for the root directory
//
auto base = communicator()->stringToProxy("RootDir:default -p 10000");

// Down-cast the proxy to a Directory proxy
//
auto rootDir = Ice::checkedCast<Filesystem::DirectoryPrx>(base);
if(!rootDir)
{
    throw "Invalid proxy";
}
```

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base
= communicator()->stringToProxy("RootDir:default -p 10000");
// Down-cast the proxy to a Directory proxy
//
Filesystem::DirectoryPrx rootDir
= Filesystem::DirectoryPrx::checkedCast(base);
if(!rootDir)
{
    throw "Invalid proxy";
}
```

For a version 2 client, the bootstrap code is almost identical — instead of down-casting to `Filesystem::Directory`, the client selects the "V2" facet during the down-cast to the type `FilesystemV2::Directory`:

C++11 C++98

```
// Create a proxy for the root directory
//
auto base = communicator()->stringToProxy("RootDir:default -p 10000");

// Down-cast the proxy to a V2 Directory proxy
//
auto rootDir = Ice::checkedCast<FilesystemV2::DirectoryPrx>(base, "V2");
if(!rootDir)
{
    throw "Invalid proxy";
}
```

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base
= communicator()->stringToProxy("RootDir:default -p 10000");

// Down-cast the proxy to a V2 Directory proxy
//
FilesystemV2::DirectoryPrx rootDir
= FilesystemV2::DirectoryPrx::checkedCast(base, "V2");
if(!rootDir)
{
    throw "Invalid proxy";
}
```

Of course, we can also create a client that can deal with both version 1 and version 2 directories: if the down-cast to version 2 fails, the client is dealing with a version 1 server and can adjust its behavior accordingly.

Behavioral Versioning

On occasion, versioning requires changes in behavior that are not manifest in the interface of the system. For example, we may have an operation that performs some work, such as:

Slice

```
interface Foo
{
    void doSomething();
}
```

The same operation on the same interface exists in both versions, but the *behavior* of `doSomething` in version 2 differs from that in version 1. The question is, how do we best deal with such behavioral changes?

Of course, one option is to simply create a version 2 facet and to carry that facet alongside the original version 1 facet. For example:

Slice

```
module V2
{
    interface Foo  // V2 facet
    {
        void doSomething();
    }
}
```

This works fine, as far as it goes: a version 2 client asks for the "V2" facet and then calls `doSomething` to get the desired effect. Depending on your circumstances, this approach may be entirely reasonable. However, if there are such behavioral changes on several interfaces, the approach leads to a more complex type system because it duplicates each interface with such a change.

A better alternative can be to create two facets of the same type, but have the implementation of those facets differ. With this approach, both facets are of type `::Foo::doSomething`. However, the implementation of `doSomething` checks which facet was used to invoke the

request and adjusts its behavior accordingly:

C++

```
void
FooI::doSomething(const Ice::Current& current)
{
    if(current.facet == "V2")
    {
        // Provide version 2 behavior...
    }
    else
    {
        // Provide version 1 behavior...
    }
}
```

This approach avoids creating separate types for the different behaviors, but has the disadvantage that version 1 and version 2 objects are no longer distinguishable to the type system. This can matter if, for example, an operation accepts a `Foo` proxy as a parameter. Let us assume that we also have an interface `FooProcessor` as follows:

Slice

```
interface FooProcessor
{
    void processFoo(Foo* w);
}
```

If `FooProcessor` also exists as a version 1 and version 2 facet, we must deal with the question of what should happen if a version 1 `Foo` proxy is passed to a version 2 `processFoo` operation because, at the type level, there is nothing to prevent this from happening.

You have two options for dealing with this situation:

- Define working semantics for mixed-version invocations. In this case, you must come up with sensible system behavior for all possible combinations of versions.
- If some of the combinations are disallowed (such as passing a version 1 `Foo` proxy to a version 2 `processFoo` operation), you can detect the version mismatch in the server by looking at the `Current::facet` member and throwing an exception to indicate a version mismatch. Simultaneously, write your clients to ensure they only pass a permissible version to `processFoo`. Clients can ensure this by checking the facet name of a proxy before passing it to `processFoo` and, if there is a version mismatch, changing either the `Foo` proxy or the `FooProcessor` proxy to a matching facet:

C++11 C++98

```
shared_ptr<FooPrx> foo = ...;                // Get a Foo...
shared_ptr<FooProcessorPrx> fooP = ...;      // Get a FooProcessor...

string fooFacet = foo->ice_getFacet();
string fooPFacet = fooP->ice_getFacet();
if(fooFacet != fooPFacet)
{
    if(fooPFacet == "V2")
    {
        error("Cannot pass a V1 Foo to a V2 FooProcessor");
    }
    else
    {
        // Upgrade FooProcessor from V1 to V2
        fooP = Ice::checkdCast<FooProcessorPrx>(fooP, "V2");
        if(!fooP)
        {
            error("FooProcessor does not have a V2 facet");
        }
        else
        {
            fooP->processFoo(foo);
        }
    }
}
```

```
FooPrx foo = ...;               // Get a Foo...
FooProcessorPrx fooP = ...;     // Get a FooProcessor...

string fooFacet = foo->ice_getFacet();
string fooPFacet = fooP->ice_getFacet();
if(fooFacet != fooPFacet)
{
    if(fooPFacet == "V2")
    {
        error("Cannot pass a V1 Foo to a V2 FooProcessor");
    }
    else
    {
        // Upgrade FooProcessor from V1 to V2
        fooP = FooProcessorPrx::checkedCast(fooP, "V2");
        if(!fooP)
        {
            error("FooProcessor does not have a V2 facet");
        }
        else
        {
            fooP->processFoo(foo);
        }
    }
}
```

Facets Design Considerations

Facets allow you to add versioning to a system, but they are merely a mechanism, not a solution. You still have to make a decision as to how to version something. For example, eventually you may want to deprecate a previous version's behavior; at that point, you must make a decision how to handle requests for the deprecated version. For behavioral changes, you have to decide whether to use separate interfaces or use facets with the same interface. And, of course, you must have compatibility rules to determine what should happen if, for example, a version 1 object is passed to an operation that implements version 2 behavior. In other words, facets cannot do your thinking for you and are no panacea for the versioning problem.

The biggest advantage of facets is also the biggest drawback: facets delay the decision about the types that are used and their behavior until run time. While this provides a lot of flexibility, it is significantly less type-safe than having explicit types that can be statically checked at compile time: if you have a problem relating to incorrect facet selection, the problem will be visible only at run time and, moreover, will be visible only if you actually execute the code that contains the problem, and execute it with just the right data.

Another danger of facets is the opportunity for abuse. As an extreme example, here is an interface that provides an arbitrary collection of objects of arbitrary type:

Slice

```
interface Collection {}
```

Even though this interface is empty, it can provide access to an unlimited number of objects of arbitrary type in the form of facets. While this example is extreme, it illustrates the design tension that is created by facets: you must decide, for a given versioning problem, how and at what point of the type hierarchy to split off a facet that deals with the changed functionality. The temptation may be to "simply add another facet" and be done with it. However, if you do that, your objects are in danger of being nothing more than loose conglomerates of facets without rhyme or reason, and with little visibility of their relationships in the type system.

In object modeling terms, the relationship among facets is weaker than an *is-a* relationship (because facets are often not type-compatible among each other). On the other hand, the relationship among facets is stronger than a *has-a* relationship (because all facets of an Ice object share the same object identity).

It is probably best to treat the relationship of a facet to its Ice object with the same respect as an inheritance relationship: if you were omniscient and could have designed your system for all current and future versions simultaneously, many of the operations that end up on separate facets would probably have been in the same interface instead. In other words, adding a facet to an Ice object most often implies that the facet has an *is-partly-a* relationship with its Ice object. In particular, if you think about the life cycle of an Ice object and find that, when an Ice object is deleted, all its facets must be deleted, this is a strong indication of a correct design. On the other hand, if you find that, at various times during an Ice object's life cycle, it has a varying number of facets of varying types, that is a good indication that you are using facets incorrectly.

Ultimately, the decision comes down to deciding whether the trade-off of static type safety versus dynamic type safety is worth the convenience and backward compatibility. The answer depends strongly on the design of your system and individual requirements, so we can only give broad advice here. Finally, there will be a point where no amount of facet trickery will get past the point when "yet one more version will be the straw that breaks the camel's back." At that point, it is time to stop supporting older versions and to redesign the system.

See Also

- Facets
- The Current Object
- Object Identity

Transports

In Ice terminology, a *transport* is a conduit for exchanging Ice protocol messages. Ice supports a number of transports; some of them are built into the Ice core, while others are available as plug-ins. Your transport selection will be driven by application requirements. For example, the TCP transport might be acceptable for deployment on a trusted intranet, while an Internet-facing application will generally use SSL or secure WebSocket.

Most transports have a configuration component, which is often done statically via the application's external configuration file. Some transports also provide an API for accessing special features programmatically. Finally, certain Ice features might not be available with all transports. For example, you can send oneway invocations with any transport, whereas twoway invocations require a stream-oriented transport and datagram invocations require the UDP transport.

You'll need to choose a transport as soon as you're ready to test your first client-server prototype, if not sooner. The server's object adapter a nd the client's proxy must have at least one matching endpoint, where an endpoint is simply a transport name together with any necessary options. Ice uses a simple text-based syntax for configuring endpoints.

All of Ice's IP-based transports support both IPv4 and IPv6, as long as the underlying platform also supports both. You can set configuration properties to enable or disable them, and control whether IPv4 or IPv6 has priority.

The following table summarizes the transports that Ice provides and indicates whether they are built into the Ice core:

Transport	Core?	Connection Oriented?	Description
TCP	Yes	Yes	This is the default transport in Ice.
UDP	Yes	No	Supports unicast and multicast datagram invocations.
SSL	No	Yes	Install the IceSSL plug-in to use a platform's native SSL implementation.
WebSocket	Yes	Yes	Especially useful when a client needs to communicate with a back-end service via a web server. IceSSL must also be installed in order to use the secure version of this transport.
Bluetooth	No	Yes	Install the IceBT plug-in to use Bluetooth on Linux and Android. IceSSL must also be installed in order to use the secure version of this transport.
iAP	No	Yes	Install the IceIAP plug-in to communicate via the Apple iAP protocol reserved for accessories. IceSSL must also be installed in order to use the secure version of this transport.

Administration and Diagnostics

This section presents the facilities that Ice provides to administer, monitor and troubleshoot Ice-based applications.

Topics

Administrative Facility

Ice applications often require remote administration, such as when an IceGrid node needs to gracefully deactivate a running server. The Ice run time provides an extensible, centralized facility for exporting administrative functionality.

Topics

See Also

The admin Object

On this page:

Overview of the admin Object

When the Administrative Facility is enabled, you can configure Ice to host an administrative object in the `Ice.Admin` object adapter, or you can programmatically host this object in your own object adapter.

In this manual, we refer to the administrative object as the *`admin` object*.

You can retrieve a proxy to the admin object associated with your communicator, if any, by calling `getAdmin` on this communicator, for example:

C++11 C++98

```
auto adminProxy = myCommunicator->getAdmin(); // adminProxy is a
std::shared_ptr<Ice::ObjectPrx>
```

```
Ice::ObjectPrx adminProxy = myCommunicator->getAdmin();
```

Facets of the admin Object

An Ice object is actually a collection of sub-objects known as facets whose types are not necessarily related. Although facets are typically used for extending and versioning types, they also allow a group of interfaces with a common purpose to be consolidated into a single Ice object with an established interface for navigation. These qualities make facets an excellent match for the requirements of the administrative facility.

Each facet of the `admin` object represents a distinct administrative capability. The object does not have a default facet (that is, a facet with an empty name). However, the Ice run time implements several built-in facets that it adds to the `admin` object:

- the `Process` facet
- the `Properties` facet
- the Logger facet
- the `Metrics` facet

An application can control which facets are installed with a configuration property. An application can also install its own facets if necessary. Administrative facets are not required to inherit from a common Slice interface.

See Also

- Creating the admin Object
- Object Identity
- Versioning
- The Process Facet
- The Properties Facet
- Filtering Administrative Facets
- Custom Administrative Facets
- Ice.Admin.*

Creating the admin Object

The administrative facility is disabled by default. To enable it, you must set the property `Ice.Admin.Enabled` to a numeric value greater than 0, or leave `Ice.Admin.Enabled` unset and specify endpoints for the `Ice.Admin` administrative object adapter using the property `Ice.Admin.Endpoints`. When `Ice.Admin.Enabled` is set to 0 or a negative value, the administrative facility is disabled, and `Ice.Admin.Endpoints` is ignored.

When the administrative facility is enabled, and

- The `Ice.Admin.Endpoints` property specifies endpoints for the Ice.Admin object adapter, and
- The `Ice.Admin.DelayCreation` property is not set, or is set to 0 or a negative value

Ice creates automatically the admin object during communicator initialization, and hosts all its enabled facets in the built-in `Ice.Admin` object adapter.

This admin object and its facets are created at the end of communicator initialization, after the initialization of all plugins. Ice gives this object the identity *`instance-name`*/admin, where *instance-name* is the value of the the `Ice.Admin.InstanceName` property. If `Ice.Admin.InstanceName` is not set or empty, Ice generates a UUID for *instance-name*.

If Ice does not create the admin object during communicator initialization as described above, you need to create the admin object after communicator initialization by calling `getAdmin` or `createAdmin` on the communicator:

Slice

```
local interface Communicator
{
    // ...
    Object* getAdmin();
    Object* createAdmin(ObjectAdapter adminAdapter, Identity adminId);
    // ...
}
```

`getAdmin` is typically used to create the admin object when both `Ice.Admin.Enpoints` and `Ice.Admin.DelayCreation` are set. The resulting admin object's enabled facets are hosted in the `Ice.Admin` object adapter. The identity of this admin object is `instance-name`/admin, where *instance-name* corresponds to the value of `Ice.Admin.InstanceName` or a UUID, as described above.

`createAdmin` is used to create the admin object and host all its enabled facets in the object adapter of your choice (the `adminAdapter` parameter), with the identity of your choice (the `adminId` parameter). If `adminAdapter` is null and `Ice.Admin.Endpoints` specifies endpoints for the `Ice.Admin` object adapter, `createAdmin` creates and activates the `Ice.Admin` object adapter and hosts all the admin object's facets in that adapter. Note that only one admin object can be created for a given communicator. `createAdmin` will fail and raise an exception if there is already an admin object.

> The administrative facility introduces additional security considerations, therefore the endpoints for the object adapter where the admin object's facets are hosted must be chosen with caution.

See Also

- Ice.Admin.*
- IceGrid and the Administrative Facility
- Security Considerations for Administrative Facets
- Using the admin Object

Using the admin Object

A program can obtain a proxy for its `admin` object by calling the `getAdmin` operation on a communicator:

Slice

```
module Ice
{
    local interface Communicator
    {
        // ...
        Object* getAdmin();
    }
}
```

This operation returns a null proxy if the administrative facility is disabled or the application has not created the admin object with `createAdmin`. The proxy returned by `getAdmin` cannot be used for invoking operations because it refers to the default facet and, as we mentioned previously, the `admin` object does not support a default facet. A program must first obtain a new version of the proxy that is configured with the name of a particular administrative facet before invoking operations on it. Although it cannot be used for invocations, the original proxy is still useful because it contains the endpoints of the object adapter where the admin object is hosted, and therefore the program may elect to export that proxy to a remote client.

To administer a program remotely, somehow you must obtain a proxy for the program's `admin` object. There are several ways for the administrative client to accomplish this:

- Construct the proxy itself, assuming that it knows the admin object's identity, facets, and endpoints. The format of the stringified proxy is as follows:
  adminId -f *admin-facet*:*admin-endpoints*
  The identity, represented here by *adminId*, is *instance-name*/admin for admin objects created during communicator initialization or with `getAdmin`, where *instance-name* is the value of the `Ice.Admin.InstanceName` property or a UUID if that property is not defined. (Clearly, the use of a UUID makes the proxy much more difficult for a client to construct on its own.) The name of the administrative facet is supplied as the value of the -f option, and the endpoints of the object adapter that hosts the admin object appear last in the proxy.
- Invoke an application-specific interface for retrieving the `admin` object's proxy.
- Use the `getServerAdmin` operation on the `IceGrid::Admin` interface, if the remote program was activated by IceGrid (see IceGrid Server Activation).

Having obtained the proxy, the administrative client must select a facet before invoking any operations. For example, the code below shows how to obtain the configuration properties of the remote program:

C++11 C++98

```
std::shared_ptr<Ice::ObjectPrx> adminObj = ...;
auto propAdmin = Ice::checkedCast<PropertiesAdminPrx>(adminObj,
"Properties");
auto props = propAdmin->getPropertiesForPrefix("");
```

```
Ice::ObjectPrx adminObj = ...;
Ice::PropertiesAdminPrx propAdmin =
Ice::PropertiesAdminPrx::checkedCast(adminObj, "Properties");
Ice::PropertyDict props = propAdmin->getPropertiesForPrefix("");
```

Here we used an overloaded version of `checkedCast` to supply the facet name of interest (`Properties`). We could have selected the facet using the proxy method `ice_facet` instead, as shown below:

C++11 C++98

```
std::shared_ptr<Ice::ObjectPrx> adminObj = ...;
auto propAdmin =
Ice::checkedCast<PropertiesAdminPrx>(adminObj->ice_facet("Properties"));
auto props = propAdmin->getPropertiesForPrefix("");
```

```
Ice::ObjectPrx adminObj = ...;
Ice::PropertiesAdminPrx propAdmin =
Ice::PropertiesAdminPrx::checkedCast(adminObj->ice_facet("Properties"));
Ice::PropertyDict props = propAdmin->getPropertiesForPrefix("");
```

This code is functionally equivalent to the first example.

A remote client must also know (or be able to determine) which facets are available in the target server. Typically this information is statically configured in the client, since the client must also know the interface types of any facets that it uses. If an invocation on a facet raises `FacetNotExistException`, the client may have used an incorrect facet name, or the server may have disabled the facet in question.

See Also

- The admin Object
- Creating the admin Object
- Ice.Admin.*
- Proxy and Endpoint Syntax
- Automatic Retries
- Versioning

The Process Facet

An activation service, such as an IceGrid node, needs a reliable way to gracefully deactivate a server. One approach is to use a platform-specific mechanism, such as POSIX signals. This works well on POSIX platforms when the server is prepared to intercept signals and react appropriately. On Windows platforms, it works less reliably for C++ servers, and not at all for Java servers. For these reasons, the `Process` facet provides an alternative that is both portable and reliable.

> Be aware of the security considerations associated with enabling the `Process` facet.

On this page:

The `Process` Interface

The Slice interface `Ice::Process` allows an activation service to request a graceful shutdown of the program:

Slice

```
module Ice
{
    interface Process
    {
        void shutdown();
        void writeMessage(string message, int fd);
    }
}
```

When `shutdown` is invoked, the object implementing this interface is expected to initiate the termination of its process. The activation service may expect the program to terminate within a certain period of time, after which it may terminate the program abruptly.

The `writeMessage` operation allows remote clients to print a message to the program's standard output (`fd` == 1) or standard error (`fd` == 2) channels.

Obtaining the Local Process Facet

We already showed how to obtain a proxy for a remote administrative facet, but suppose you want to interact with the facet in your local address space. The code below shows the necessary steps:

C++11 C++98

```
auto obj = communicator->findAdminFacet("Process"); // obj is a
shared_ptr<Ice::Object>
if(obj)
{
    // May be null if the facet is not enabled
    auto facet = std::dynamic_pointer_cast<Ice::Process>(obj);
    ...
}
```

```
Ice::ObjectPtr obj = communicator->findAdminFacet("Process");
if(obj)
{
    // May be null if the facet is not enabled
    Ice::ProcessPtr facet = Ice::ProcessPtr::dynamicCast(obj);
    ...
}
```

As shown here, the facet is registered with the name `Process` and a regular language cast is used to downcast the base object type to the `Process` interface.

Application Requirements for the Process Facet

The default implementation of the `Process` facet requires cooperation from an application in order to successfully terminate a process. Specifically, the facet invokes `shutdown` on its communicator and assumes that the application uses this event as a signal to commence its termination procedure. For example, an application typically uses a thread (often the main thread) to call the communicator operation `waitForShutdown`, which blocks the calling thread until the communicator is shut down or destroyed. After `waitForShutdown` returns, the calling thread can initiate a graceful shutdown of its process.

Replacing the Process Facet

You can replace the default `Process` facet if your application requires a different scheme for gracefully shutting itself down. To define your own facet, create a servant that implements the `Ice::Process` interface. As an example, the servant definition shown below duplicates the functionality of the default `Process` facet:

C++11 C++98

```
class ProcessI : public Ice::Process
{
public:

    ProcessI(const shared_ptr<Ice::Communicator>& communicator)
        : _communicator(communicator)
    {
    }

    virtual void shutdown(const Ice::Current&) override
    {
        _communicator->shutdown();
    }

    virtual void writeMessage(string msg, int fd, const Ice::Current&)
override
    {
        if(fd == 1)
        {
            cout << msg << endl;
        }
        else if(fd == 2)
        {
            cerr << msg << endl;
        }
    }

private:
    const shared_ptr<Ice::Communicator> _communicator;
};
```

```
class ProcessI : public Ice::Process
{
public:

    ProcessI(const Ice::CommunicatorPtr& communicator)
        : _communicator(communicator)
    {
    }

    virtual void shutdown(const Ice::Current&)
    {
        _communicator->shutdown();
    }

    virtual void writeMessage(const string& msg, Ice::Int fd,
const Ice::Current&)
    {
        if(fd == 1)
        {
            cout << msg << endl;
        }
        else if(fd == 2)
        {
            cerr << msg << endl;
        }
    }

private:
    const Ice::CommunicatorPtr _communicator;
};
```

As you can see, the default implementation of `shutdown` simply shuts down the communicator, which initiates an orderly termination of the Ice run time's server-side components and prevents object adapters from dispatching any new requests. You can add your own application-specific behavior to the `shutdown` method to ensure that your program terminates in a timely manner.

> A servant must not invoke `destroy` on its communicator while executing a dispatched operation.

To avoid the risk of a race condition, the recommended strategy for replacing the `Process` facet is to delay creation of the administrative facets until after communicator initialization, so that your application has a chance to replace the facet:

```
# Delay admin object creation for admin object hosted in the Ice.Admin
object adapter
Ice.Admin.DelayCreation=1
```

With `Ice.Admin.DelayCreation` enabled, the application can safely remove the default `Process` facet and install its own:

C++11 C++98

```
shared_ptr<Ice::Communicator> communicator = ...
communicator->removeAdminFacet("Process");
auto myProcessFacet = make_shared<MyProcessFacet>(...);
communicator->addAdminFacet(myProcessFacet, "Process");
```

```
Ice::CommunicatorPtr communicator = ...
communicator->removeAdminFacet("Process");
Ice::ProcessPtr myProcessFacet = new MyProcessFacet(...);
communicator->addAdminFacet(myProcessFacet, "Process");
```

If you host the admin object in the `Ice.Admin` object adapter, the final step is to create the admin object by calling `getAdmin` on the communicator:

C++

```
communicator->getAdmin();
```

And if you host the admin object in your own object adapter, the final set is to create the admin object with `createAdmin`:

C++

```
communicator->createAdmin(myAdapter, myAdminId);
```

Integrating the Process Facet with an Activation Service

If the `Ice.Admin.ServerId` and `Ice.Default.Locator` properties are defined, the Ice run time performs the following steps after creating the admin object:

- Obtains proxies for the `Process` facet and the default locator
- Invokes `getRegistry` on the proxy to obtain a proxy for the locator registry
- Invokes `setServerProcessProxy` on the locator registry and supplies the value of `Ice.Admin.ServerId` along with a proxy for the `Process` facet

The identifier specified by `Ice.Admin.ServerId` must uniquely identify the process within the locator registry.

In the case of IceGrid, IceGrid defines the `Ice.Admin.ServerId` and `Ice.Default.Locator` properties for each deployed server. IceGrid also supplies a value for `Ice.Admin.Endpoints` if neither this property nor Ice.Admin.Enabled are defined by the server.

See Also

- Communicator
- Versioning
- Security Considerations for Administrative Facets
- Object Adapters
- Creating the admin Object
- Ice.Admin.*
- IceGrid

The Properties Facet

On this page:

The `PropertiesAdmin` Interface

An administrator may find it useful to be able to view or modify the configuration properties of a remote Ice application. For example, the Ice Grid administrative tools allow you to query and update the properties of active servers. The `Properties` facet supplies this functionality.

The `Ice::PropertiesAdmin` interface provides access to the communicator's configuration properties:

Slice

```
module Ice
{
    interface PropertiesAdmin
    {
        string getProperty(string key);
        PropertyDict getPropertiesForPrefix(string prefix);
        void setProperties(PropertyDict newProperties);
    }
}
```

The `getProperty` operation retrieves the value of a single property, and the `getPropertiesForPrefix` operation returns a dictionary of properties whose keys match the given prefix. These operations have the same semantics as those in the `Ice::Properties` interface.

The `setProperties` operation merges the entries in `newProperties` with the communicator's existing properties. If an entry in `newProperties` matches the name of an existing property, that property's value is replaced with the new value. If the new value is an empty string, the property is removed. Any existing properties that are not modified or removed by the entries in `newProperties` are retained with their original values. If the `Ice.Trace.Admin.Properties` property is enabled, Ice logs a message if a call to `setProperties` results in any changes to the property set.

> Modifying a program's configuration properties at run time may not have an effect on the program. For example, many of Ice's standard configuration properties are read once during communicator initialization, and never again.

Obtaining the Local Properties Facet

We already showed how to obtain a proxy for a remote administrative facet, but suppose you want to interact with the facet in your local address space. The code below shows the necessary steps:

C++11 Java

```
auto obj = communicator->findAdminFacet("Properties");
if(obj)  // May be null if the facet is not enabled
{
    auto facet = std::dynamic_pointer_cast<Ice::PropertiesAdmin>(obj);
    ...
}
```

```
com.zeroc.Ice.Object obj = communicator.findAdminFacet("Properties");
if(obj != null)  // May be null if the facet is not enabled
{
     facet = (com.zeroc.Ice.PropertiesAdmin)obj;
    ...
}
```

As shown here, the facet is registered with the name `Properties` and a regular language cast is used to downcast the base object type to the `PropertiesAdmin` interface.

Property Update Notifications

The Ice run time can notify an application whenever its properties change due to invocations of the `setProperties` operation on the `PropertiesAdmin` interface. This section describes the native API for receiving property updates.

C++11 C++98 C# Java Java Compat

The `Properties` facet object provided by the Ice run time derives from the class `NativePropertiesAdmin`:

```
namespace Ice
{
    class NativePropertiesAdmin
    {
    public:

        virtual ~NativePropertiesAdmin();
        virtual std::function<void()>
addUpdateCallback(std::function<void(const PropertyDict&)>) = 0;
    };
}
```

`addUpdateCallback` accepts a function parameter (the update callback) and returns another function. Call this returned function to remove the callback.

The callback receives a `dictionary<string, string>` value representing the properties that were added, changed or removed, with removed properties denoted by an entry whose value is an empty string. It is legal for the callback implementation to modify the set of callbacks, and Ice ignores any exceptions it might raise.

The following code demonstrates how to register a callback:

```
auto obj = communicator->findAdminFacet("Properties");
if(obj)
{
    auto facet =
std::dynamic_pointer_cast<Ice::NativePropertiesAdmin>(obj);
    auto propManager = ...
    facet->addUpdateCallback([propManager](const Ice::PropertyDict&
changes) { propManager->updated(changes); });
}
```

The `Properties` facet object provided by the Ice run time derives from the class `NativePropertiesAdmin`:

```
namespace Ice
{
    class NativePropertiesAdmin : public virtual IceUtil::Shared
    {
    public:

        virtual ~NativePropertiesAdmin();
        virtual void addUpdateCallback(const
PropertiesAdminUpdateCallbackPtr&) = 0;
        virtual void removeUpdateCallback(const
PropertiesAdminUpdateCallbackPtr&) = 0;
    };
    typedef IceUtil::Handle<NativePropertiesAdmin>
NativePropertiesAdminPtr;
}
```

The application must supply an instance of `PropertiesAdminUpdateCallback`:

```
namespace Ice
{
    class PropertiesAdminUpdateCallback : virtual public
Ice::LocalObject
    {
    public:
        virtual void updated(const PropertyDict& changes) = 0;
    };
    typedef IceUtil::Handle<PropertiesAdminUpdateCallback>
PropertiesAdminUpdateCallbackPtr;
}
```

The `updated` method receives a `dictionary<string, string>` value representing the properties that were added, changed or removed, with removed properties denoted by an entry whose value is an empty string. It is legal for the `updated` implementation to modify the set of callbacks, and Ice ignores any exceptions it might raise.

The following code demonstrates how to register a callback:

```
Ice::ObjectPtr obj = communicator->findAdminFacet("Properties");
if(obj)
{
    Ice::NativePropertiesAdminPtr facet =
Ice::NativePropertiesAdminPtr::dynamicCast(obj);
    Ice::PropertiesAdminUpdateCallbackPtr myCallback = new ...;
    facet->addUpdateCallback(myCallback);
}
```

The `Properties` facet object provided by the Ice run time derives from the class `NativePropertiesAdmin`:

```
namespace Ice
{
    public interface NativePropertiesAdmin
    {
        void addUpdateCallback(System.Action<Dictionary<string, string>>
callback);
        void removeUpdateCallback(System.Action<Dictionary<string,
string>> callback);
    }
}
```

The registered callback method receives a `dictionary<string, string>` value representing the properties that were added, changed or removed, with removed properties denoted by an entry whose value is an empty string. It is legal for the callback implementation to modify the set of callbacks, and Ice ignores any exceptions it might raise.

The following code demonstrates how to register a callback:

```
Ice.Object obj = communicator.findAdminFacet("Properties");
if(obj != null)
{
    Ice.NativePropertiesAdmin facet = obj as Ice.NativePropertiesAdmin;
    propManager = ...;
    facet.addUpdateCallback(propManager.updated);
}
```

The `Properties` facet object provided by the Ice run time derives from the class `NativePropertiesAdmin`:

```
package com.zeroc.Ice;

public interface NativePropertiesAdmin
{
    void
addUpdateCallback(java.util.function.Consumer<java.util.Map<String,
String>> callback);
    void
removeUpdateCallback(java.util.function.Consumer<java.util.Map<String,
String>> callback);
}
```

The registered callback method receives a `dictionary<string, string>` value representing the properties that were added, changed or removed, with removed properties denoted by an entry whose value is an empty string. It is legal for the callback implementation to modify the set of callbacks, and Ice ignores any exceptions it might raise.

The following code demonstrates how to register a callback:

```
com.zeroc.Ice.Object obj = communicator.findAdminFacet("Properties");
if(obj != null)
{
    com.zeroc.Ice.NativePropertiesAdmin facet =
(com.zeroc.Ice.NativePropertiesAdmin)obj;
    propManager = ...
    facet.addUpdateCallback(changes -> propManager.updated(changes));
}
```

The `Properties` facet object provided by the Ice run time derives from the class `NativePropertiesAdmin`:

```
package Ice;

public interface PropertiesAdminUpdateCallback
{
    void updated(java.util.Map<String, String> changes);
}

public interface NativePropertiesAdmin
{
    void addUpdateCallback(PropertiesAdminUpdateCallback callback);
    void removeUpdateCallback(PropertiesAdminUpdateCallback callback);
}
```

The registered callback method receives a `dictionary<string, string>` value representing the properties that were added, changed or removed, with removed properties denoted by an entry whose value is an empty string. It is legal for the callback implementation to modify the set of callbacks, and Ice ignores any exceptions it might raise.

The following code demonstrates how to register a callback:

```
Ice.Object obj = communicator.findAdminFacet("Properties");
if(obj != null)
{
    Ice.NativePropertiesAdmin facet = (Ice.NativePropertiesAdmin)obj;
    Ice.PropertiesAdminUpdateCallback callback = ...
    facet.addUpdateCallback(callback);
}
```

See Also

- Properties and Configuration
- Versioning
- IceGrid

The Logger Facet

On this page:

The `RemoteLogger` Interface

An administrator may find it useful to view the log of a running Ice application, without going through an intermediary file. This is especially useful for Ice services that use a system log, such as the Windows Event Log.

The `Logger` facet allows remote applications (such as administrative clients) to attach one or more remote loggers to the local logger of any Ice application (provided this application enables the admin object with a `Logger` facet). The implementation of the `Logger` facet installs its own logger, which intercepts the log messages sent to the local logger, caches the most recent log messages, and forwards these log messages (after optional filtering) to the attached remote loggers.

A remote logger is an Ice object that implements the `Ice::RemoteLogger` interface. Such object is typically implemented by an administrative client.

Slice

```
module Ice
{
    enum LogMessageType { PrintMessage, TraceMessage, WarningMessage,
ErrorMessage }
    sequence<LogMessageType> LogMessageTypeSeq;

    struct LogMessage
    {
       LogMessageType type;
       long timestamp;
       string traceCategory;
       string message;
    }
    sequence<LogMessage> LogMessageSeq;

    interface RemoteLogger
    {
        void init(string prefix, LogMessageSeq logMessages);
        void log(LogMessage message);
    }
}
```

The `LogMessage` `struct` represents log messages sent to a local logger. Its `timestamp` data member is the number of microseconds since the Unix Epoch (January 1st, 1970 at 0:00 UTC).

When a remote logger is attached to a local logger, its `init` operation is called with the local logger's prefix and a list of recent log messages (see The LoggerAdmin Interface below). Then, each time a log message is sent to the local logger, the Logger facet forwards this message to the remote logger's `log` operation.

> The Logger facet does not guarantee that `init` will be called on a remote logger before `log` is called on this remote logger, even though the log messages sent to `init` are always older than the log messages sent to `log`. It is indeed common for `log` to be called several times before `init` in applications that generate many logs. An implementation of `RemoteLogger` needs to handle

this situation correctly: it can for example keep all log messages received before `init` in a queue, and later append this queue to the log messages received through `init`.

The `LoggerAdmin` Interface

The `Logger` facet implements the `Ice::LoggerAdmin` interface:

Slice

```
module Ice
{
    interface LoggerAdmin
    {
        void attachRemoteLogger(RemoteLogger* prx, LogMessageTypeSeq
messageTypes, StringSeq traceCategories, int messageMax)
            throws RemoteLoggerAlreadyAttachedException;

        bool detachRemoteLogger(RemoteLogger* prx);

        LogMessageSeq getLog(LogMessageTypeSeq messageTypes, StringSeq
traceCategories, int messageMax, out string prefix);
    }
}
```

The operation `attachRemoteLogger` attaches a remote logger (`prx`) to the local logger with the following optional filters:

- `messageTypes` specifies the types of log messages this remote logger wants to receive. An empty sequence means no filtering–the `Logger` facet will forward all log message types.
- `traceCategories` is a sequence of categories for trace messages. An empty sequence means no filtering–the `Logger` facet will forward all trace categories.
- `messageMax` is the maximum number of log messages sent to init. If `messageMax` is negative, all available log messages will be sent to `init` (provided they satisfy the `messageTypes` and `traceCategories` filters). If messageMax is 0, no log message will be sent to `init`. If `messageMax` is greater than 0, the most recent `messageMax` log messages that satisfy the `messageTypes` and `traceCategories` filters will be sent to `init`.

For example, you can attach a remote logger with no filtering at all as follows:

C++11 C++98

```
std::shared_ptr<Ice::RemoteLoggerPrx> remoteLogger = // my remote
logger, typically an object implemented by the local application
std::shared_ptr<Ice::ObjectPrx> admin = // get proxy to the admin object
for the target application
auto loggerAdmin = Ice::uncheckedCast<Ice::LoggerAdminPrx>(admin,
"Logger");
loggerAdmin->attachRemoteLogger(remoteLogger, Ice::LogMessageTypeSeq(),
Ice::StringSeq(), -1);
```

```
Ice::RemoteLoggerPrx remoteLogger = // my remote logger, typically an
object implemented by the local application
Ice::ObjectPrx admin = // get proxy to the admin object for the target
application
Ice::LoggerAdminPrx loggerAdmin =
Ice::LoggerAdminPrx::uncheckedCast(admin, "Logger");
loggerAdmin->attachRemoteLogger(remoteLogger, Ice::LogMessageTypeSeq(),
Ice::StringSeq(), -1);
```

If you are interested only in errors and trace messages for category `Network` with no more than 10 such messages sent to init, you could write instead:

C++11 C++98

```
std::shared_ptr<Ice::RemoteLoggerPrx> remoteLogger = // my remote
logger, typically an object implemented by the local application
std::shared_ptr<Ice::ObjectPrx> admin = // get proxy to the admin object
for the target application
auto loggerAdmin = Ice::uncheckedCast<Ice::LoggerAdminPrx>(admin,
"Logger");

Ice::LogMessageTypeSeq messageTypes;
messageTypes.push_back(Ice::LogMessageType::ErrorMessage);
messageTypes.push_back(Ice::LogMessageType::TraceMessage);

Ice::StringSeq traceCategories;
traceCategories.push_back("Network");

loggerAdmin->attachRemoteLogger(remoteLogger, messageTypes,
traceCategories, 10);
```

```
Ice::RemoteLoggerPrx remoteLogger = // my remote logger, typically an
object implemented by the local application
Ice::ObjectPrx admin = // get proxy to the admin object for the target
application
Ice::LoggerAdminPrx loggerAdmin =
Ice::LoggerAdminPrx::uncheckedCast(admin, "Logger");

Ice::LogMessageTypeSeq messageTypes;
messageTypes.push_back(Ice::ErrorMessage);
messageTypes.push_back(Ice::TraceMessage);

Ice::StringSeq traceCategories;
traceCategories.push_back("Network");

loggerAdmin->attachRemoteLogger(remoteLogger, messageTypes,
traceCategories, 10);
```

The operation `detachRemoteLogger` detaches a remote logger (`prx`) from the local logger. `prx` does not need to match exactly the proxy provided to `attachRemoteLogger`: the `Logger` facet uses only the identity in this proxy. `detachRemoteLogger` returns true if this remote logger was found (and is now detached), and false otherwise.

The Logger facet automatically detaches a remote logger when a request sent to this remote logger fails. As a result, if an administrative client forgets to call `detachRemoteLogger` after destroying its remote logger, or crashes, this remote logger will remain attached only as long as it's not used.

The operation `getLog` retrieves the most recent `messageMax` log messages that match the optional `messageTypes` and `traceCategories` filters. These parameters have the same meaning as in the `attachRemoteLogger` operation described above. For example, if you want to retrieve only the 10 most recent errors and trace messages for category `Network`, you could write:

C++11 C++98

```
std::shared_ptr<Ice::ObjectPrx> admin = // get proxy to the admin object
for the target application
auto loggerAdmin = Ice::uncheckedCast<Ice::LoggerAdminPrx>(admin,
"Logger");

Ice::LogMessageTypeSeq messageTypes;
messageTypes.push_back(Ice::LogMessageType::ErrorMessage);
messageTypes.push_back(Ice::LogMessageType::TraceMessage);

Ice::StringSeq traceCategories;
traceCategories.push_back("Network");

string prefix;
auto logMessages = loggerAdmin->getLog(messageTypes, traceCategories,
10, prefix);
```

```
Ice::ObjectPrx admin = // get proxy to the admin object for the target
application
Ice::LoggerAdminPrx loggerAdmin =
Ice::LoggerAdminPrx::uncheckedCast(admin, "Logger");

Ice::LogMessageTypeSeq messageTypes;
messageTypes.push_back(Ice::ErrorMessage);
messageTypes.push_back(Ice::TraceMessage);

Ice::StringSeq traceCategories;
traceCategories.push_back("Network");

string prefix;
Ice::LogMessageSeq logMessages = loggerAdmin->getLog(messageTypes,
traceCategories, 10, prefix);
```

Configuring the Logger Facet

The `Logger` facet caches the most recent log messages sent to application's Logger, to be able to provide these log messages to remote loggers (in the `init` operation) and to administrative clients that call `getLog` (see the `LoggerAdmin` Interface above). You can configure how many log messages are cached by the Logger facet with the `Ice.Admin.Logger.KeepLogs` and `Ice.Admin.Logger.KeepTraces` properties. The default is to keep the most recent 100 log messages other than trace messages plus the most recent 100 trace messages.

See Also

- Logger Facility

The Metrics Facet

The `Metrics` facet implements the Instrumentation Facility to provide convenient access to metrics for the Ice run time and select Ice services. The metrics provided by this facet include the number of threads currently running and their state, the number of connections, information on invocations and dispatch, as well as connection establishment and endpoint name resolution.

On this page:

Metrics Terminology

- metric: an "analytical measurement intended to quantify the state of a system", recorded by the Ice run time, such as bytes sent over a connection.
- metric name: the name of the metric, such as "number of connections".
- metrics map: a collection of metrics objects.
- metrics view: a collection of metrics maps. A view contains metrics maps of different types (e.g., connections and threads). Several metrics views can be configured with different purposes. For example, you can have a "debug" metrics view to get detailed metrics of each of the instrumented objects in the Ice communicator. This view can be enabled from time to time for debugging purposes but it's disabled most of the time. You could also have a more coarse-grained metrics view to collect data at a higher level, such as the amount of bytes received and sent by all the connections from the communicator. This metrics view can be enabled all the time.

Metrics Types

Metrics are specified as Slice classes defined in the `Ice/Metrics.ice` Slice file. All the metrics types are defined in the `IceMX` module.

The base class is `IceMX::Metrics`:

Slice

```
class Metrics
{
    string id;
    long total = 0;
    int current = 0;
    long totalLifetime = 0;
    int failures = 0;
}
```

A metrics object is an instance of `IceMX::Metrics` and represents metrics of one or more instrumented objects. An instrumented object can be anything that supports instrumentation. The Ice run time supports instrumentation of the following objects and activities:

- Threads
- Connections
- Invocations
- Dispatch
- Connection establishment
- Endpoint resolution

The `id` of a metric identifies the instrumented object(s). The `total` and `current` members specify the total and current number of instrumented objects created since the creation of the Ice communicator, respectively. The `totalLifetime` member is the sum of the lifetime of each instrumented object and `failures` is the number of failures that have occurred for the metrics object(s).

Failures are specified using a separate `IceMX::MetricsFailures` structure:

Slice

```
struct MetricsFailures
{
    string id;
    StringIntDict failures;
}
```

The `failures` dictionary provides the count of each type of failure for a metric identified by `id`. Failures are dependent on the instrumented objects. For example, failures for Ice connections are represented with the name of the exception that caused the connection to fail (e.g., `Ice::ConnectionLostException` or `Ice::TimeoutException`).

A metrics map is simply defined as a sequence of `IceMX::Metrics` objects, and a metrics view is defined as a dictionary of metrics map:

Slice

```
sequence<Metrics> MetricsMap;
dictionary<string, MetricsMap> MetricsView;
```

The key for the metrics view dictionary is a string that identifies the metrics map. The Ice run time supports the following metrics maps:

Metrics map name	Slice class	Description
Connection	`IceMX::ConnectionMetrics`	Connection metrics.
Thread	`IceMX::ThreadMetrics`	Thread metrics. The Ice run time instruments threads for the communicator's thread pools as well as threads used internally.
Invocation	`IceMX::InvocationMetrics`	Client-side proxy invocation metrics.
Dispatch	`IceMX::DispatchMetrics`	Server-side dispatch metrics.
EndpointLookup	`IceMX::Metrics`	Endpoint lookup metrics. For tcp, ssl and udp endpoints, this corresponds to the DNS lookups made to resolve the host names in endpoints.
ConnectionEstablishment	`IceMX::Metrics`	Connection establishment metrics.

A metrics map can also contain sub-metrics maps. An example is the `Invocation` metrics map, which provides a `Remote` sub-metrics map to record metrics associated with remote invocations. The Slice class for remote invocation metrics is `IceMX::Metrics`.

The `MetricsAdmin` Interface

The Slice interface `IceMX::MetricsAdmin` allows you to retrieve the metrics associated with the Ice communicator:

Slice

```
module IceMX
{
    exception UnknownMetricsView {}

    interface MetricsAdmin
    {
        Ice::StringSeq getMetricsViewNames(out Ice::StringSeq
disableViews);
        void enableMetricsView(string name)
            throws UnknownMetricsView;
        void disableMetricsView(string name)
            throws UnknownMetricsView;
        MetricsView getMetricsView(string view, out long timestamp)
            throws UnknownMetricsView;
        MetricsFailuresSeq getMapMetricsFailures(string view, string
map)
            throws UnknownMetricsView;
        MetricsFailures getMetricsFailures(string view, string map,
string id)
            throws UnknownMetricsView;
    }
}
```

The `getMetricsViewName` operation retrieves the names of the configured enabled and disabled views. The enableMetricsView and disableMetricsView allow to enable and disable a specific view. Calling those methods is equivalent to setting the view Disabled property to 1 or 0.The `getMetricsView` operation returns the metrics for the given view. The `getMapMetricsFailures` and `getMetricsFailures` operations retrieve the metrics failures for a given map or metrics id.

Obtaining the Local Metrics Facet

We already showed how to obtain a proxy for a remote administrative facet, but suppose you want to interact with the facet in your local address space. The code below shows the necessary steps:

C++11 C++98

```
auto obj = communicator->findAdminFacet("Metrics");
if(obj)
{
    // May be null if the facet is not enabled
    auto facet = std::dynamic_pointer_cast<Ice::MetricsAdmin>(obj);
    ...
}
```

```
Ice::ObjectPtr obj = communicator->findAdminFacet("Metrics");
if(obj)
{
    // May be null if the facet is not enabled
    Ice::MetricsAdminPtr facet = Ice::MetricsAdminPtr::dynamicCast(obj);
    ...
}
```

As shown here, the facet is registered with the name `Metrics` and a regular language cast is used to downcast the base object type to the `MetricsAdmin` interface.

Metrics Attributes

Metrics views are configured with IceMX Metrics properties.

The `GroupBy`, `Accept` and `Reject` properties are specified using attributes that are specific to each metrics map. The table below describes the attributes supported by the Ice run time's metrics maps.

Name	Maps	Description
id	All	A unique identifier to select the instrumented object or operation.
parent	All	The parent of the instrumented object or operation.
none	All	The none attribute is a special attribute that evaluates to the empty string.
endpoint	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	The stringified endpoint.
endpointType	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	The endpoint numerical type as defined in `Ice/Endpoint.ice`.
endpointIsDatagram	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	A boolean indicating if the endpoint is a datagram endpoint.
endpointIsSecure	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	A boolean indicating if the endpoint is secure.
endpointTimeout	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	The endpoint timeout.
endpointCompress	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	A boolean indicating if the endpoint requires compression.
endpointHost	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	The endpoint host.
endpointPort	Connection, Dispatch, Remote, ConnectionEstablishment, EndpointLookup	The endpoint port.
connection	Dispatch	The connection description.
incoming	Connection, Dispatch, Remote	A boolean where true indicates an incoming (server) connection and false an outgoing (client) connection.
adapterName	Connection, Dispatch, Remote	If the connection is a server connection, adapterName returns the name of the adapter that created the connection, otherwise it is the empty string.

connectionId	Connection, Dispatch, Remote	The ID of the connection if one is set, otherwise it is the empty string.
localAddress	Connection, Dispatch, Remote	The connection's local address.
localPort	Connection, Dispatch, Remote	The connection's local port.
remoteAddress	Connection, Dispatch, Remote	The connection's remote address.
remotePort	Connection, Dispatch, Remote	The connection's remote port.
mcastAddress	Connection, Dispatch, Remote	The connection's multicast address.
mcastPort	Connection, Dispatch, Remote	The connection's multicast port.
state	Connection	The state of the connection.
operation	Dispatch, Invocation	The dispatched or invoked operation name.
identity	Dispatch, Invocation	The identity of the Ice object used for the dispatch or invocation.
facet	Dispatch, Invocation	The facet of the Ice object used for the dispatch or invocation.
mode	Dispatch, Invocation	The dispatch or invocation mode.
context.*key*	Dispatch, Invocation	The value of the dispatch or invocation context with the given key.
proxy	Invocation	The proxy used for the invocation.
encoding	Invocation	The proxy encoding.

The `id`, `parent` and `none` attributes are supported by all maps.

The value of the `parent` attribute depends on the map. For the Connection, Dispatch and Remote maps, the `parent` will either be "Communicator" if the connection is a client connection, or the object adapter name if it's a server connection. For the Invocation, EndpointLookup, and ConnectionEstablishment maps, it will always be "Communicator". The `parent` attribute enables the filtering of metrics based on the object adapter. When used with the `GroupBy` property it also allows you to obtain metrics at the object adapter level. For instance, the following configuration does not monitor any metrics for the `Ice.Admin` object adapter and it groups all the metrics based on the object adapter or communicator:

```
IceMX.Metrics.MyView.GroupBy=parent
IceMX.Metrics.MyView.Reject.parent=Ice\.Admin   # Escape the dot in
Ice.Admin
```

This configuration enables the communicator to get metrics on a per object adapter or communicator basis.

You can also use the `none` attribute to get metrics for the communicator including the metrics from object adapters, e.g., `IceMX.Metrics.MyView.GroupBy=none`. This provides the lowest possible level of detail as all the statistics will be recorded by a single metrics object.

The `id` attribute allows you to get a higher level of detail by recording metrics on a per instrumented object or operation basis. If you specify `IceMX.Metrics.MyView.GroupBy=id`, the `Metrics` facet will record metrics for each individual object or operation.

See Also

- Instrumentation Facility
- Creating the admin Object
- Ice.Admin.*

Filtering Administrative Facets

The Ice run time enables all of its built-in administrative facets by default, and an application may install its own custom facets. You can control which facets the Ice run time enables using the `Ice.Admin.Facets` property. For example, the following property definition enables the `Properties` facet and leaves the `Process` facet (and any application-defined facets) disabled:

```
Ice.Admin.Facets=Properties
```

To specify more than one facet, separate them with a comma or white space. A facet whose name contains white space must be enclosed in single or double quotes.

> The Ice run time creates only the built-in administrative facets that are enabled. Disabled built-in facets are not created at all. In Ice releases prior to Ice 3.6, the built-in facets were always created, whether enabled or disabled through `Ice.Admin.Facets`.

See Also

- The Process Facet
- The Properties Facet
- The Metrics Facet
- Custom Administrative Facets
- Ice.Admin.*

Custom Administrative Facets

An application can add and remove administrative facets using the `Communicator` operations shown below:

Slice

```
module Ice
{
    local interface Communicator
    {
        // ...
        void addAdminFacet(Object servant, string facet);
        Object removeAdminFacet(string facet);
    }
}
```

The `addAdminFacet` operation installs a new facet with the given name, or raises `AlreadyRegisteredException` if a facet already exists with the same name. The `removeAdminFacet` operation removes (and returns) the facet with the given name, or raises `NotRegisteredException` if no matching facet is found.

The mechanism for filtering administrative facets also applies to application-defined facets. If you call `addAdminFacet` while a filter is in effect, and the name of your custom facet does not match the filter, the Ice run time will not expose your facet but instead keeps a reference to it so that a subsequent call to `removeAdminFacet` is possible.

We provide an example of using these communicator operations in our discussion of the `Process` facet.

See Also

- Filtering Administrative Facets
- The Process Facet

Security Considerations for Administrative Facets

Exposing administrative functionality naturally makes a program vulnerable, therefore it is important that proper precautions are taken. With respect to the default functionality, the `Properties` facet could expose sensitive configuration information, and the `Process` facet supports a `shutdown` operation that opens the door for a denial-of-service attack. Developers should carefully consider the security implications of any additional administrative facets that an application installs.

There are several approaches you can take to mitigate the possibility of abuse:

- Disable the administrative facility

  The administrative facility is disabled by default, and remains disabled as long as its prerequisites are not met. Note that IceGrid enables the facility in servers that it activates for the following reasons:
  - The `Process` facet allows the IceGrid node to gracefully terminate the process.
  - The `Properties` facet enables IceGrid administrative clients to obtain configuration information about activated servers.
  - The `Logger` facet enables IceGrid administrative clients to attach remote loggers to the Loggers of activated servers.

  You could disable a facet using filtering, but doing so may disrupt IceGrid's normal operation.

- Select a proper endpoint

  If you host the admin object in the `Ice.Admin` object adapter, a reasonably secure value for the `Ice.Admin.Endpoints` property is one that uses the local host interface (`-h 127.0.0.1`), which restricts access to clients that run on the same host. Incidentally, this is the default value that IceGrid defines for its servers, although you can override that if you like. Note that using a local host endpoint does not preclude remote administration for IceGrid servers because IceGrid transparently routes requests on `admin` objects to the appropriate server via its node. If your application must support administration from non-local hosts, we recommend the use of SSL and certificate-based access control.

- Filter the facets

  After choosing a suitable endpoint, you can minimize risks by filtering the facets to enable only the functionality that is required. For example, if you are not using IceGrid's server activation feature and do not require the ability to remotely terminate a program, you should disable the `Process` facet using the filtering mechanism.

- Consider the object's identity

  The default identity of an `admin` object created during communicator initialization (or with `getAdmin`) has a UUID for its category, which makes it difficult for a hostile client to guess. Depending on your requirements, the use of a UUID may be an advantage or a disadvantage. For example, in a trusted environment, the use of a UUID may create additional work, such as the need to add an interface that an administrative client can use to obtain the identity or proxy of a remote `admin` object. An obscure identity might be more of a hindrance in this situation, and therefore specifying a static category via the `Ice.Admin.InstanceName` property is a reasonable alternative. In general, however, we recommend using the default behavior.

See Also

- The admin Object
- The Properties Facet
- The Process Facet
- Creating the admin Object
- IceGrid and the Administrative Facility
- Filtering Administrative Facets
- Ice.Admin.*
- IceGrid
- IceSSL

Logger Facility

Depending on the setting of various properties, the Ice run time produces trace, warning, or error messages. These messages are written via the `Ice::Logger` interface:

Slice

```
module Ice
{
    local interface Logger
    {
        void print(string message);
        void trace(string category, string message);
        void warning(string message);
        void error(string message);
        string getPrefix();
        Logger cloneWithPrefix(string prefix);
    }
}
```

The `cloneWithPrefix` operation returns a new logger that logs to the same destination but with a different prefix. (The prefix is used to, for example, provide the name of the process writing the log messages.)

Topics

- The Default Logger
- Custom Loggers
- Built-in Loggers
- Logger Plug-ins
- The Per-Process Logger
- C++ Logger Utility Classes

See Also

- Properties and Configuration

The Default Logger

A default logger is instantiated when you create a communicator. The default logger writes its messages to the standard error output. The `trace` operation accepts a `category` parameter in addition to the error message; this allows you to separate trace output from different subsystems by sending the output through a filter.

You can obtain the logger that is attached to a communicator using the `getLogger` operation:

Slice

```
module Ice
{
    local interface Communicator
    {
        Logger getLogger();
        // ...
    }
}
```

On Windows, when the default C++ logger outputs a log message to the console, it converts this message from your narrow string encoding (as defined by narrow string converter you installed, if any) to your console code page. This conversion is disabled if you redirect standard error to a file with `Ice.StdErr`, or if you set `Ice.LogStdErr.Convert` to 0.

See Also

- Logger Facility

Custom Loggers

You have several options if you wish to install a logger other than the default one:

- Select one of the other built-in loggers, which allow you to log to a file, to the `syslog` on Unix, and to the Windows event log
- Supply your own logger implementation in an `InitializationData` parameter when you create a communicator
- Load a logger implementation dynamically via the Ice plug-in facility

Changing the `Logger` object that is attached to a communicator allows you to integrate Ice messages into your own message handling system. For example, for a complex application, you might have an existing logging framework. To integrate Ice messages into that framework, you can create your own `Logger` implementation that logs messages to the existing framework.

When you destroy a communicator, its logger is *not* destroyed. This means that you can safely use a logger even beyond the lifetime of its communicator.

See Also

- Built-in Loggers
- Communicator Initialization
- Logger Plug-ins

Built-in Loggers

Ice provides a file-based logger as well as Unix- and Windows-specific logger implementations. For .NET, the default Ice logger uses a `TraceListener` and so can be customized at run time via configuration.

On this page:

File Logger

The file-based logger is enabled via the `Ice.LogFile` property. This logger is available for all supported languages and platforms.

Syslog Logger

You can activate a logger that logs via the Unix `syslog` implementation by setting the `Ice.UseSyslog` property. This logger is available in Ice for C++, Java, and C#, as well as for scripting languages based on Ice for C++.

Windows Logger

On Windows, subclasses of `Ice::Service` use the Windows application event log by default. The event log implementation is available for C++ applications.

.NET Logger

The default logger in Ice for .NET writes its messages using the `System.Diagnostics.Trace` facility. By default, the Ice run time registers a `ConsoleTraceListener` that writes to `stderr`. You can disable the logging of messages via this trace listener by setting the property `Ice.ConsoleListener` to zero.

You can change the trace listener for your application via the application's configuration file. For example:

```
<configuration>
  <system.diagnostics>
    <trace autoflush="false" indentsize="4">
      <listeners>
        <add name="myListener"
          type="System.Diagnostics.EventLogTraceListener"
          initializeData="TraceListenerLog" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

This configuration installs a trace listener that logs to the Windows event log using the source name `TraceListenerLog`.

> The `EventLogTraceListener` creates a new event source if no match is found for the source name defined by `initializeData`, and creating a new event source requires that you run the application with administrative privileges. Alternatively, you can create the event source in advance using an administrative tool. For more information, search MSDN for "EventLog Component".

See Also

- Service Helper Class

Logger Plug-ins

Installing a custom logger using the Ice plug-in facility has several advantages. Because the logger plug-in is specified by a configuration property and loaded dynamically by the Ice run time, an application requires no code changes in order to utilize a custom logger implementation. Furthermore, a logger plug-in takes precedence over the per-process logger and the logger supplied in the `InitializationData` argument during communicator initialization, meaning you can use a logger plug-in to override the logger that an application installs by default.

On this page:

Installing a C++ Logger Plug-in

To install a logger plug-in in C++, you must first define a subclass of `Ice::Logger`:

C++11 C++98

```
class MyLoggerI : public Ice::Logger
{
public:

    virtual void print(const std::string& message) override;
    virtual void trace(const std::string& category,
const std::string& message) override;
    virtual void warning(const std::string& message) override;
    virtual void error(const std::string& message) override;
    virtual std::string getPrefix() override;
    virtual std::shared_ptr<Ice::Logger> cloneWithPrefix(const string&
prefix) override;

    // ...
};
```

```
class MyLoggerI : public Ice::Logger
{
public:

    virtual void print(const std::string& message);
    virtual void trace(const std::string& category,
const std::string& message);
    virtual void warning(const std::string& message);
    virtual void error(const std::string& message);
    virtual std::string getPrefix();
    virtual LoggerPtr cloneWithPrefix(const string& prefix);

    // ...
};
```

Next, supply a factory function that installs your custom logger by returning an instance of `Ice::LoggerPlugin`:

C++11 C++98

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
    createLogger(const std::shared_ptr<Ice::Communicator>& communicator,
                 const std::string& name,
                 const Ice::StringSeq& args)
    {
        auto logger = std::make_shared<MyLoggerI>();
        return new Ice::LoggerPlugin(communicator, logger);
    }
}
```

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
    createLogger(const Ice::CommunicatorPtr& communicator,
                 const std::string& name,
                 const Ice::StringSeq& args)
    {
        Ice::LoggerPtr logger = new MyLoggerI;
        return new Ice::LoggerPlugin(communicator, logger);
    }
}
```

The factory function can have any name you wish; we used `createLogger` in this example. Refer to the plug-in API for more information on plug-in factories.

The definition of `LoggerPlugin` is shown below:

C++11 C++98

```
namespace Ice
{
    class LoggerPlugin : public Plugin
    {
    public:

 LoggerPlugin(const std::shared_ptr<Communicator>&, const std::shared_p
tr<Logger>&);

        virtual void initialize() override;
        virtual void destroy() override;
    }
}
```

```
namespace Ice
{
    class LoggerPlugin : public Plugin
    {
    public:
        LoggerPlugin(const CommunicatorPtr&, const LoggerPtr&);

        virtual void initialize();
        virtual void destroy();
    }
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=mylogger:createLogger
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property represents the plug-in's entry point, in which `mylogger` is the abbreviated form of its shared library or DLL, and `createLogger` is the name of the factory function.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.MyLogger.cpp=mylogger:createLogger
```

Installing a Java Logger Plug-in

To install a logger plug-in in Java, you must first define a subclass of `Logger`:

Java

```
public class MyLoggerI implements com.zeroc.Ice.Logger
{
    public void print(String message) { ... }
    public void trace(String category, String message) { ... }
    public void warning(String message) { ... }
    public void error(String message) { ... }
    public String getPrefix() { ... }
    public Logger cloneWithPrefix(String prefix) { ... }

    // ...
}
```

Next, define a factory class that installs your custom logger by returning an instance of `LoggerPlugin`:

Java

```
public class MyLoggerPluginFactoryI implements com.zeroc.Ice.PluginFact
ory
{
    public com.zeroc.Ice.Plugin create(com.zeroc.Ice.Communicator commu
nicator, String name, String[] args)
    {
        com.zeroc.Ice.Logger logger = new MyLoggerI();
        return new com.zeroc.Ice.LoggerPlugin(communicator, logger);
    }
}
```

The factory class can have any name you wish; in this example, we used `MyLoggerPluginFactoryI`. Refer to the plug-in API for more information on plug-in factories.

The definition of `LoggerPlugin` is shown below:

Java Java Compat

```
package com.zeroc.Ice;

public class LoggerPlugin implements Plugin
{
    public LoggerPlugin(Communicator communicator, Logger logger) { ...
}

    public void initialize() { }

    public void destroy() { }
}
```

```
package Ice;

public class LoggerPlugin implements Plugin
{
    public LoggerPlugin(Communicator communicator, Logger logger) { ...
}

    public void initialize() { }

    public void destroy() { }
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=MyLoggerPluginFactoryI
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property is the name of the factory class.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for Java run time:

```
Ice.Plugin.MyLogger.java=MyLoggerPluginFactoryI
```

Installing a C# Logger Plug-in

To install a logger plug-in in .NET, you must first define a subclass of `Ice.Logger`:

C#

```
public class MyLoggerI : Ice.Logger
{
    public void print(string message) { ... }
    public void trace(string category, string message) { ... }
    public void warning(string message) { ... }
    public void error(string message) { ... }
    public string getPrefix() { ... }
    public Logger cloneWithPrefix(string prefix) { ... }

    // ...
}
```

Next, define a factory class that installs your custom logger by returning an instance of `Ice.LoggerPlugin`:

C#

```
public class MyLoggerPluginFactoryI : Ice.PluginFactory
{
    public Ice.Plugin create(Ice.Communicator communicator,
string name, string[] args)
    {
        Ice.Logger logger = new MyLoggerI();
        return new Ice.LoggerPlugin(communicator, logger);
    }
}
```

The factory class can have any name you wish; in this example, we used `MyLoggerPluginFactoryI`. Refer to the plug-in API for more information on plug-in factories. Typically the logger implementation and the factory are compiled into a single assembly.

The definition of `LoggerPlugin` is shown below:

C#

```
namespace Ice
{
    public partial class LoggerPlugin : Plugin
    {
        public LoggerPlugin(Communicator communicator, Logger logger)
        {
            // ...
        }

        public void initialize() { }

        public void destroy() { }
    }
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=mylogger.dll:MyLoggerPluginFactoryI
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property is the entry point for the factory, consisting of an assembly name followed by the name of the factory class.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for .NET run time:

```
Ice.Plugin.MyLogger.clr=mylogger.dll:MyLoggerPluginFactoryI
```

See Also

- Custom Loggers
- The Per-Process Logger
- Communicator Initialization
- Plug-in API
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

The Per-Process Logger

Ice allows you to install a per-process custom logger. This logger is used by all communicators that do not have their own specific logger established at the time a communicator is created.

You can set a per-process logger in C++ by calling `Ice::setProcessLogger`, and you can retrieve the per-process logger by calling `Ice::getProcessLogger`:

C++11 C++98

```
std::shared_ptr<Ice::Logger> getProcessLogger();
void setProcessLogger(const std::shared_ptr<Logger>&);
```

```
LoggerPtr getProcessLogger();
void setProcessLogger(const LoggerPtr&);
```

If you call `getProcessLogger` without having called `setProcessLogger` first, the Ice run time installs a default per-process logger. Note that if you call `setProcessLogger`, only communicators created after that point will use this per-process logger; communicators created earlier use the logger that was in effect at the time they were created. (This also means that you can call `setProcessLogger` multiple times; communicators created after that point will use whatever logger was established by the last call to `setProcessLogger`.)

`getProcessLogger` and `setProcessLogger` are language-specific APIs that are not defined in Slice. Therefore, these methods appear in the `com.zeroc.Ice.Util` class (for Java), and the `Ice.Util` class (for Java Compat and C#).

For applications that use the `Application` or `Service` convenience classes and do not explicitly configure a logger, these classes set a default per-process logger that uses the `Ice.ProgramName` property as a prefix for log messages.

See Also

- Custom Loggers
- Communicator Initialization
- Application Helper Class
- Service Helper Class

C++ Logger Utility Classes

The Ice run time supplies a collection of utility classes that make use of the logger facility simpler and more convenient. Each of the `Logger` interface's four operations has a corresponding helper class:

C++11 C++98

```
namespace Ice
{
    class Print
    {
    public:
        Print(const std::shared_ptr<Logger>&);
        void flush();
        ...
    }

    class Trace
    {
    public:
        Trace(const std::shared_ptr<Logger>&, const std::string&);
        void flush();
        ...
    }

    class Warning
    {
    public:
        Warning(const std::shared_ptr<Logger>&);
        void flush();
        ...
    }

    class Error
    {
    public:
        Error(const std::shared_ptr<Logger>&);
        void flush();
        ...
    }
}
```

```
namespace Ice
{
    class Print
    {
    public:
        Print(const LoggerPtr&);
        void flush();
        ...
    }

    class Trace
    {
    public:
        Trace(const LoggerPtr&, const std::string&);
        void flush();
        ...
    }

    class Warning
    {
    public:
        Warning(const LoggerPtr&);
        void flush();
        ...
    }

    class Error
    {
    public:
        Error(const LoggerPtr&);
        void flush();
        ...
    }
}
```

The only notable difference among these classes is the extra argument to the `Trace` constructor; this argument represents the trace category.

To use one of the helper classes in your application, you simply instantiate it and compose your message:

C++

```
if(errorCondition)
{
    Error err(communicator->getLogger());
    err << "encountered error condition: " << errorCondition;
}
```

The Ice run time defines the necessary stream insertion operators so that you can treat an instance of a helper class as if it were a standard

C++ output stream. When the helper object is destroyed, its destructor logs the message you have composed. If you want to log more than one message using the same helper object, invoke the `flush` method on the object to log what you have composed so far and reset the object for a new message.

The helper classes also supply insertion operators to simplify the task of logging an exception. The operators accept instances of `std::exception` (from which all Ice exceptions derive) and log the string returned by the `what` method. On some platforms, you can also enable the configuration property `Ice.PrintStackTraces`, which causes the helper classes to log the stack trace of the exception in addition to the value of `what`.

See Also

- Logger Facility

Instrumentation Facility

The Ice run time can be instrumented using observer interfaces that monitor many aspects of the run time's internal objects and activities, including connections, threads, servant dispatching, proxy invocations, endpoint lookups and connection establishment. We refer to these internal objects and activities as "instrumented objects" in the discussion below. The application is responsible for implementing the instrumentation observer interfaces. Note however that an implementation of these interfaces is provided by the Metrics Facet, so most applications do not need to implement them and can instead collect metrics through the `MetricsAdmin` facet. The definition of the instrumentation interfaces can be found in the `Ice/Instrumentation.ice` Slice file.

The Ice run time uses the `Ice::Instrumentation::CommunicatorObserver` interface to obtain observers for instrumented objects created by the run time:

Slice

```
module Ice
{
    module Instrumentation
    {
        local interface CommunicatorObserver
        {
            Observer getConnectionEstablishmentObserver(Endpoint endpt,
string connector);
            Observer getEndpointLookupObserver(Endpoint endpt);
            ConnectionObserver getConnectionObserver(ConnectionInfo c,
Endpoint e,
                                                     ConnectionState s,
ConnectionObserver o);
            ThreadObserver getThreadObserver(string parent, string id,
ThreadState s,
                                             ThreadObserver o);
            InvocationObserver getInvocationObserver(Object* prx, string
operation, Context ctx);
            DispatchObserver getDispatchObserver(Current c);
            void setObserverUpdater(ObserverUpdater updater);
        }
    }

    local interface Communicator
    {
        Ice::Instrumentation::CommunicatorObserver getObserver();
        // ...
    }
}
```

The Ice run time calls the appropriate `get...Observer` operation each time a new instrumented object is created. The implementation of these methods should return an observer, or nil if the implementation does not want to monitor the instrumented object. This observer is associated with the instrumented object and receives notifications of any changes to its attributes or state. All observer interfaces derive from the `Ice::Instrumentation::Observer` base interface:

Slice

```
local interface Observer
{
    void attach();
    void detach();
    void failed(string exceptionName);
}
```

The `attach` operation is called upon association of the observer with the new instrumented object. The `detach` operation is called when the object is destroyed. The `failed` operation is called to report any failures that might occur during the lifetime of the instrumented object. Observer specializations provide additional operations for monitoring other attributes. For example, here is the `Ice::Instrumentation::ConnectionObserver` interface:

Slice

```
local interface ConnectionObserver extends Observer
{
    void sentBytes(int num);
    void receivedBytes(int num);
}
```

The `sentBytes` and `receivedBytes` methods are called by the Ice connection when new bytes are received or sent over the connection.

Shown below is an implementation of the communicator and connection observer interfaces to record sent and received bytes on a per-connection basis. The observer dumps how many bytes were received and sent for the connection when it is detached:

C++11 C++98

```
class ConnectionObserverImpl : public
Ice::Instrumentation::ConnectionObserver
{
public:
    ConnectionObserverImpl(const shared_ptr<Ice::ConnectionInfo>&
connInfo)
        : info(connInfo), sentBytes(0), receivedBytes(0)
    {
    }

    virtual void attach() override
    {
    }

    virtual void detach() override
    {
        cerr << info->remoteHost << ":" << info->remotePort << ": sent
bytes = "
             << sentBytes << ", received bytes = " << receivedBytes <<
endl;
    }
```

```
    virtual void sentBytes(int num) override
    {
        sentBytes += num;
    }

    virtual void receivedBytes(int num) override
    {
        receivedBytes += num;
    }

    virtual void failed(const std::string&) override
    {
    }

private:

    shared_ptr<Ice::ConnectionInfo> info;
    long sentBytes;
    long receivedBytes;
};

class CommunicatorObserverImpl : public
Ice::Instrumentation::CommunicatorObserver
{
public:

    virtual shared_ptr<Ice::Instrumentation::ConnectionObserver>
    getConnectionObserver(const shared_ptr<Ice::ConnectionInfo>& c,
const shared_ptr<Ice::Endpoint>& e,
                          Ice::Instrumentation::ConnectionState s,
                          const
shared_ptr<Ice::Instrumentation::ConnectionObserver>& previous) override
    {
```

```
        return make_shared<ConnectionObserverImpl>(c);
    }
};
```

```
class ConnectionObserverImpl : public
Ice::Instrumentation::ConnectionObserver
{
public:
    ConnectionObserverImpl(const Ice::ConnectionInfoPtr& connInfo)
        : info(connInfo), sentBytes(0), receivedBytes(0)
    {
    }

    void attach()
    {
    }

    void detach()
    {
        cerr << info->remoteHost << ":" << info->remotePort << ": sent
bytes = "
             << sentBytes << ", received bytes = " << receivedBytes <<
endl;
    }

    void sentBytes(int num)
    {
        sentBytes += num;
    }

    void receivedBytes(int num)
    {
        receivedBytes += num;
    }

    void failed(const std::string&)
    {
    }

private:

    Ice::ConnectionInfoPtr info;
    long sentBytes;
    long receivedBytes;
};

class CommunicatorObserverImpl : public
```

```
Ice::Instrumentation::CommunicatorObserver
{
public:

    Ice::Instrumentation::ConnectionObserverPtr
    getConnectionObserver(const Ice::ConnectionInfoPtr& c, const
Ice::EndpointPtr& e,
                          Ice::Instrumentation::ConnectionState s,
                          const
Ice::Instrumentation::ConnectionObserverPtr& previous)
    {
```

```
        return new ConnectionObserverImpl(c);
    }
};
```

For brevity we have omitted the implementation of the other `get...Observer` methods; they all return 0 as we are only interested in instrumenting connections.

To register your implementation, you must pass it in an `InitializationData` parameter when you initialize a communicator:

C++11 C++98

```
Ice::InitializationData initData;
id.observer = std::make_shared<CommunicatorObserverImpl>();
Ice::CommunicatorHolder ich(initData);
```

```
Ice::InitializationData initData;
id.observer = new CommunicatorObserverImpl;
Ice::CommunicatorHolder ich(initData);
```

You can install a `CommunicatorObserver` object on either the client or the server side (or both). Here is some example output produced by installing our `CommunicatorObserver` and `ConnectionObserver` object implementations in a simple server:

```
127.0.0.1:3487: sent bytes = 14, received bytes = 32
127.0.0.1:3487: sent bytes = 33, received bytes = 14
127.0.0.1:3490: sent bytes = 14, received bytes = 14
...
```

In addition to the operations for retrieving observers, the `CommunicatorObserver` interface also defines a `setObserverUpdater` operation that is called by the Ice run time on initialization to provide an updater object to the `CommunicatorObserver` implementation. This updater object can be used to "refresh" some of the created observers. The updater object provided by the Ice run time implements the following interface:

Slice

```
local interface ObserverUpdater
{
    void updateConnectionObservers();
    void updateThreadObservers();
}
```

The `CommunicatorObserver` implementation can call these operations to update the observers associated with Ice connections or threads. When one of these operations is called, the Ice run time calls the matching `get...Observer` method on the `CommunicatorObserver` interface for each of the instrumented objects. For example, if you call `updateConnectionObservers`, your implementation of `getConnectionObserver` will be called again for each Ice connection in the communicator. The `previous` parameter to `getConnectionObserver` represents the observer that is currently associated with the connection.

This mechanism can be used to re-configure the observers associated with instrumented objects. For instance, the application might not wish to instrument connections all the time but only when needed. It can use the observer updater to enable or disable the instrumentation. Here is the example above modified to provide this functionality:

C++11C++98

```
class CommunicatorObserverImpl : public
Ice::Instrumentation::CommunicatorObserver
{
public:

    virtual shared_ptr<Ice::Instrumentation::ConnectionObserver>
    getConnectionObserver(const shared_ptr<Ice::ConnectionInfo>& c,
const shared_ptr<Ice::Endpoint>& e,
                          Ice::Instrumentation::ConnectionState s,
                          const
shared_ptr<Ice::Instrumentation::ConnectionObserver>& previous) override
    {
        lock_guard<mutex> lk(_mutex);
        return enabled ? make_shared<ConnectionObserverImpl>(c) :
nullptr;
    }

    virtual void setEnabled(bool enabled) override
    {
        {
            lock_guard<mutex> lk(_mutex);
            if(this->enabled == enabled)
            {
                return;
            }
            this->enabled = enabled;
        }
        updater->updateConnections();
    }

    virtual void setObserverUpdater(const
shared_ptr<Ice::Instrumentation::ObserverUpdater>& updater) override
    {
        this->updater = updater;
    }

    const shared_ptr<Ice::Instrumentation::ObserverUpdater> updater;
    bool enabled;
    std::mutex _mutex;
};
```

```
class CommunicatorObserverImpl : public
Ice::Instrumentation::CommunicatorObserver
{
public:

    Ice::Instrumentation::ConnectionObserverPtr
    getConnectionObserver(const Ice::ConnectionInfoPtr& c, const
Ice::EndpointPtr& e,
                          Ice::Instrumentation::ConnectionState s,
                          const
Ice::Instrumentation::ConnectionObserverPtr& previous)
    {
        Lock sync(_mutex);
        return enabled ? new ConnectionObserverImpl(c) : 0;
    }

    void setEnabled(bool enabled)
    {
        {
            Lock sync(_mutex);
            if(this->enabled == enabled)
            {
                return;
            }
            this->enabled = enabled;
        }
        updater->updateConnections();
    }

    void setObserverUpdater(const
Ice::Instrumentation::ObserverUpdaterPtr& updater)
    {
        this->updater = updater;
    }

    const Ice::Instrumentation::ObserverUpdaterPtr updater;
    bool enabled;
    IceUtil::Mutex _mutex;
};
```

As you can see in the example above, special care needs to be taken with respect to synchronization. The Ice run time can call observers with Ice internal locks held to guarantee consistency of the information passed to the `get...Observer` methods. It is therefore important that the implementation of your observers performs quickly and does not create deadlocks. Your observers should not make remote invocations or call Ice APIs that require acquiring locks on instrumented objects.

See Also

- Communicator Initialization
- The Metrics Facet

IceBox

IceBox is an easy-to-use framework for Ice application services. The Service Configurator pattern [1] is a useful technique for configuring services and centralizing their administration. In practical terms, this means services are developed as dynamically-loadable components that can be configured into a general purpose "super server" in whatever combinations are necessary. IceBox is an implementation of the Service Configurator pattern for Ice services.

A generic IceBox server replaces the typical monolithic Ice server you normally write. The IceBox server is configured via properties with the application-specific services it is responsible for loading and managing, and it can be administered remotely. There are several advantages in using this architecture:

- Services loaded by the same IceBox server can be configured to take advantage of Ice's collocation optimizations. For example, if one service is a client of another service, and those services reside in the same IceBox server, then invocations between them can be optimized.
- Composing an application consisting of various services is done by configuration, not by compiling and linking. This decouples the service from the server, allowing services to be combined or separated as needed.
- Multiple Java services can be active in a single instance of a Java Virtual Machine (JVM). This conserves operating system resources when compared to running several monolithic servers, each in its own JVM.
- Services implement an IceBox service interface, providing a common framework for developers and a centralized administrative facility.
- IceBox support is integrated into IceGrid, the server activation and deployment service.

IceBox offers a refreshing change of perspective: developers focus on writing services, not applications. The definition of an application changes as well; using IceBox, an application becomes a collection of discrete services whose composition is determined dynamically by configuration, rather than statically by the linker.

Topics

References

1. Jain, P., et al. 1997. Dynamically Configuring Communication Services with the Service Configurator Pattern. C++ Report 9 (6).

Developing IceBox Services

On this page:

The IceBox `Service` Interface

Writing an IceBox service requires implementing the IceBox `Service` interface:

Slice

```
module IceBox
{
    local interface Service
    {
        void start(string name, Ice::Communicator communicator,
Ice::StringSeq args);
        void stop();
    }
}
```

As you can see, a service needs to implement only two operations, `start` and `stop`. These operations are invoked by the server; `start` is called after the service is loaded, and `stop` is called when the IceBox server is shutting down.

The `start` operation is the service's opportunity to initialize itself; this typically includes creating an object adapter and servants. The `name` a nd `args` parameters supply information from the service's configuration, and the `communicator` parameter is an `Ice::Communicator` ob ject created by the server for use by the service. Depending on the service configuration, this communicator instance may be shared by other services in the same IceBox server, therefore care should be taken to ensure that items such as object adapters are given unique names.

The `stop` operation must reclaim any resources used by the service. Generally, a service deactivates its object adapter, and may also need to invoke `waitForDeactivate` on the object adapter in order to ensure that all pending requests have been completed before the clean up process can proceed. The server is responsible for destroying the communicator instance that was passed to `start`.

Whether the service's implementation of `stop` should explicitly destroy its object adapter depends on other factors. For example, the adapter should be destroyed if the service uses a shared communicator, especially if the service could eventually be restarted. In other circumstances, the service can allow its adapter to be destroyed as part of the communicator's destruction.

These interfaces are declared as `local` for a reason: they represent a contract between the server and the service, and are not intended to be used by remote clients. Any interaction the service has with remote clients is done via servants created by the service.

IceBox Service Example in C++

The example we present here is taken from the `IceBox/hello` sample program provided in the Ice distribution.

The class definition for our service is quite straightforward:

C++11 C++98

```
#include <IceBox/IceBox.h>

class HelloServiceI : public IceBox::Service
{
public:

    virtual void start(const std::string&, const
std::shared_ptr<Ice::Communicator>&, const Ice::StringSeq&) override;
    virtual void stop() override;

private:

    std::shared_ptr<Ice::ObjectAdapter> _adapter;
};
```

```
#include <IceBox/IceBox.h>

class HelloServiceI : public IceBox::Service
{
public:

    virtual void start(const std::string&, const Ice::CommunicatorPtr&,
const Ice::StringSeq&);
    virtual void stop();

private:

    Ice::ObjectAdapterPtr _adapter;
};
```

The member definitions are equally straightforward:

C++11 C++98

```
#include <Ice/Ice.h>
#include <HelloServiceI.h>
#include <HelloI.h>

using namespace std;

void
HelloServiceI::start(const string& name,
                     const shared_ptr<Ice::Communicator>& communicator,
                     const Ice::StringSeq& args)
{
    _adapter = communicator->createObjectAdapter(name);
    auto object = make_shared<HelloI>(communicator);
    _adapter->add(object, Ice::stringToIdentity("hello"));
    _adapter->activate();
}

void
HelloServiceI::stop()
{
    _adapter->deactivate();
}
```

```
#include <Ice/Ice.h>
#include <HelloServiceI.h>
#include <HelloI.h>

using namespace std;

void
HelloServiceI::start(const string& name,
                     const Ice::CommunicatorPtr& communicator,
                     const Ice::StringSeq& args)
{
    _adapter = communicator->createObjectAdapter(name);
    Ice::ObjectPtr object = new HelloI(communicator);
    _adapter->add(object, Ice::stringToIdentity("hello"));
    _adapter->activate();
}

void
HelloServiceI::stop()
{
    _adapter->deactivate();
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown),

and activates the object adapter. The `stop` method simply deactivates the object adapter.

C++ Service Entry Point

The last piece of the puzzle is the *entry point* function, which the IceBox server calls to obtain an instance of the service:

C++11 C++98

```
extern "C"
{
    ICE_DECLSPEC_EXPORT IceBox::Service*
    create(const shared_ptr<Ice::Communicator>&)
    {
        return new HelloServiceI;
    }
}
```

```
extern "C"
{
    ICE_DECLSPEC_EXPORT IceBox::Service*
    create(Ice::CommunicatorPtr)
    {
        return new HelloServiceI;
    }
}
```

In this example, the `create` function returns a new instance of the `Hello` service. The name of the function is not important, but it must have the signature shown above. In particular, the function must have C linkage, accept a single parameter (a `const std::shared_ptr<Ice::Communicator>&` in C++ and a `Ice::CommunicatorPtr` in C++98) and return a native pointer to `IceBox::Service`.

Configuring IceBox Services provides more information on entry points and describes how to configure your service into an IceBox server.

IceBox Service Example in Java

As with the C++ example presented in the previous section, the complete source for the Java example can be found in the `IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

Java

```
public class HelloServiceI implements com.zeroc.IceBox.Service
{
    public void start(String name,
com.zeroc.Ice.Communicator communicator, String[] args)
    {
        _adapter = communicator.createObjectAdapter(name);
        com.zeroc.Ice.Object object = new HelloI(communicator);
        _adapter.add(object, com.zeroc.Ice.Util.stringToIdentity("hello
"));
        _adapter.activate();
    }

    public void stop()
    {
        _adapter.deactivate();
    }

    private com.zeroc.Ice.ObjectAdapter _adapter;
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The server requires a service implementation to provide a public constructor (a default constructor or a constructor that accepts an Ice communicator parameter). This is the *entry point* for a Java IceBox service; that is, the server dynamically loads the service implementation class and invokes this public constructor to obtain an instance of the service.

This example is a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Configuring IceBox Services.

IceBox Service Example in C#

The complete source for the C# example can be found in the `IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

C#

```
class HelloServiceI : IceBox.Service
{
    public void start(string name, Ice.Communicator communicator,
string[] args)
    {
        _adapter = communicator.createObjectAdapter(name);
        _adapter.add(new HelloI(), Ice.Util.stringToIdentity("hello"));
        _adapter.activate();
    }

    public void stop()
    {
        _adapter.deactivate();
    }

    private Ice.ObjectAdapter _adapter;
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The server requires a service implementation to have a public constructor (a default constructor or a constructor that accepts an Ice communicator parameter). This is the entry point for a C# IceBox service; that is, the server dynamically loads the service implementation class from an assembly and invokes this public constructor to obtain an instance of the service.

This example is a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Configuring IceBox Services.

IceBox Service Failures

An exception raised by a service's implementation of its entry point, `start`, or `stop` methods causes IceBox to log a message. An exception that occurs during server startup also results in server termination.

A service implementation can indicate a failure by raising `IceBox::FailureException`:

Slice

```
module IceBox
{
    local exception FailureException
    {
        string reason;
    }
}
```

Note that, as a local exception, C++ users must instantiate `FailureException` with file and line number information:

C++

```
throw IceBox::FailureException(__FILE__, __LINE__, "my error message");
```

See Also

- Configuring IceBox Services
- Starting the IceBox Server

Configuring IceBox Services

On this page:

Installing an IceBox Service

A service is configured into an IceBox server using a single `IceBox.Service` property. This property serves several purposes: it defines the name of the service, it provides the server with the service entry point, and it defines properties and arguments for the service.

The format of the property is shown below:

```
IceBox.Service.name=entry_point [args]
```

The *name* component of the property key is the service name (IceStorm, in this example). This name is passed to the service's `start` operation, and must be unique among all services configured in the same IceBox server. It is possible, though rarely necessary, to load two or more instances of the same service under different names.

The first argument in the property value is the entry point specification. Any arguments following the entry point specification are examined. If an argument has the form `--`*`name=value`*, then it is interpreted as a property definition that appears in the property set of the communicator passed to the service `start` operation. These arguments are removed, and any remaining arguments are passed to the `start` operation in the `args` parameter.

IceBox Service Configuration in C++

For a C++ service, the entry point must have the form *library[,version]:symbol*, where *library* is the simple name of the service's shared library or DLL, and *symbol* is the name of the entry point function. A "simple name" is one without any platform-specific prefixes or extensions; the server adds appropriate decorations depending on the platform. The simple name may include a leading path, and the version is optional. If specified, the version is embedded in the library name.

As an example, here is how we could configure IceStorm, which is implemented as an IceBox service in C++:

```
IceBox.Service.IceStorm=IceStormService,37:createIceStorm
```

IceBox uses the information provided in the entry point specification to compose a library name. For the IceStorm example shown above, IceBox on Windows would compose the library name `IceStormService37.dll`. If IceBox is compiled with debug information, it appends a `d` to the library name, so the name becomes `IceStormService37d.dll` instead.

> The exact name of the library that is loaded depends on the naming conventions of the platform IceBox executes on. For example, on an OS X machine, the library name is `libIceStormService37.dylib`.

If the simple name does not include a leading path, the shared library or DLL must reside in a directory that appears in `PATH` on Windows or the shared library search path (such as `LD_LIBRARY_PATH`) on POSIX systems.

The entry point function, *symbol*, must have the signature that we originally presented in our example:

C++11 C++98

```
extern "C" IceBox::Service* function(const
shared_ptr<Ice::Communicator>&);
```

```
extern "C" IceBox::Service* function(Ice::CommunicatorPtr);
```

The communicator instance passed to this function is the IceBox server's communicator and should only be used for administrative purposes. For example, the entry point function could use this communicator's logger to display log messages. For a service's normal operations, it must use the communicator that it receives as an argument to its `start` method.

Here is a sample configuration for our C++ service:

```
IceBox.Service.Hello=HelloService:create --Ice.Trace.Network=1 hello
there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in `HelloService.dll` on Windows or `libHelloService.so` on Linux, and the entry point function `create` is invoked to create an instance of the service. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

IceBox Service Configuration in C#

The entry point of a .NET service has the form `assembly:class`. The assembly component can be a partially or fully qualified assembly name, or an assembly path name.

The details on how assemblies are load depends on how you define the assembly component and the .NET framework used by the application:

Value for *`assembly`*	Example	Semantics

Assembly name	`hello,Version=...,Culture=neutral,publicKeyToken=...`
	
	`hello`
		The assembly name can be a fully or partially qualified assembly name. The assembly is loaded using Assembly.Load. For more information on how the run-time locates assemblies, see:
		
		• how the .NET Framework runtime locates assemblies
		• how .NET Core loads assemblies.

Assembly path name	MyService.dll
	
	`services\MyService.dll`
	
	`C:\services\MyService.dll`
		The path name can be an absolute path name or a path name relative to the iceboxnet's current working directory. The assembly is loaded using Assembly.LoadFrom.

The *`class`* component is the complete class name of the service implementation class, which must define a public constructor.

To instantiate the service, the IceBox server first checks to see if the service defines a constructor taking an argument of type `Ice.Communicator`. If so, the service invokes this constructor and passes the server's communicator, which should only be used for administrative purposes. For example, the constructor could use this communicator's logger to display log messages. For a service's normal operations, it must use the communicator that it receives as an argument to its `start` method.

If the service does not define a constructor taking an `Ice.Communicator` argument, the server invokes the service's default constructor.

Here is a sample configuration for our C# example:

```
IceBox.Service.Hello=helloservice.dll:HelloServiceI
--Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the assembly named `helloservice.dll`, implemented by the class `HelloServiceI`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

IceBox Service Configuration in Java

For a Java service, the entry point is typically the class name (including any package) of the service implementation class, but may also include a leading path to a class directory or JAR file. The class must define a public constructor.

To instantiate the service, the IceBox server first checks to see if the service defines a constructor taking an argument of type `Ice.Communicator`. If so, the service invokes this constructor and passes the server's communicator, which should only be used for administrative purposes. For example, the constructor could use this communicator's logger to display log messages. For a service's normal operations, it must use the communicator that it receives as an argument to its `start` method.

If the service does not define a constructor taking an `Ice.Communicator` argument, the server invokes the service's default constructor.

Here is a sample configuration for our Java example:

```
IceBox.Service.Hello=HelloServiceI --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the class `HelloServiceI`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Load Order for IceBox Services

By default, the server loads the configured services in an undefined order, meaning services in the same IceBox server should not depend on one another. If services must be loaded in a particular order, the `IceBox.LoadOrder` property can be used:

```
IceBox.LoadOrder=Service1,Service2
```

In this example, `Service1` is loaded first, followed by `Service2`. Any remaining services are loaded after `Service2`, in an undefined order. Each service mentioned in `IceBox.LoadOrder` must have a matching `IceBox.Service` property.

During shutdown, services are stopped in the reverse of the order in which they were loaded.

Using a Shared Communicator

IceBox creates a separate communicator instance for each service by default in order to minimize the chances of accidental conflicts among services. You can optionally specify that certain services use a shared communicator instead by setting `IceBox.UseSharedCommunicator.`*`name`* properties in the server's configuration:

```
IceBox.Service.Hello=...
IceBox.Service.Printer=...
IceBox.UseSharedCommunicator.Hello=1
IceBox.UseSharedCommunicator.Printer=1
```

A common use case for sharing a communicator between two or more services is enabling the use of collocation optimizations for invocations among those services. This optimization is not possible with the default behavior that creates a new communicator for each service.

IceBox prepares the property set of this shared communicator as follows:

- If services inherit the server's properties, the property set initially contains the server's properties (excluding `Ice.Admin.Endpoints`), otherwise the property set starts out empty.
- For each service that uses the shared communicator:
  - Merge its properties into the shared property set, overwriting any existing properties with the same names
  - Remove any properties from the shared property set that the service explicitly clears. For example, the definition `Ice.Trace.Network=` clears any existing setting of `Ice.Trace.Network` in the shared property set.
  - Translate service-specific command-line settings into properties (e.g., `--Hello.Debug=1`)

Service properties are merged in the same order as the services are loaded. As a result, the final value of a property that is defined by multiple services depends on the order in which those services are loaded. Let's expand our example to demonstrate this behavior:

```
# File: server.cfg
IceBox.Service.Hello=... --Ice.Config=hello.cfg --Hello.Debug=1
IceBox.Service.Printer=... --Ice.Config=printer.cfg
IceBox.UseSharedCommunicator.Hello=1
IceBox.UseSharedCommunicator.Printer=1
IceBox.InheritProperties=1
IceBox.LoadOrder=Hello,Printer
Ice.Trace.Network=1

# File: hello.cfg
Ice.Trace.Network=2

# File: printer.cfg
Ice.Trace.Network=3
```

The two services `Hello` and `Printer` use the shared communicator, and all services inherit the server's properties. As IceBox prepares the shared property set, the initial value of `Ice.Trace.Network` is `1` as defined in the (inherited) server's configuration. The `IceBox.LoadOrder` property specifies that the `Hello` service should be loaded first, therefore the `Ice.Trace.Network` property momentarily has the value `2` but ultimately has the value `3` after the properties for the `Printer` service are merged.

If we change the value of `IceBox.LoadOrder` so that IceBox loads `Printer` first, the value for `Ice.Trace.Network` in the shared communicator will be `2` instead because the setting in `hello.cfg` overrides all previous values.

Inheriting Properties from the IceBox Server

By default, a service does not inherit the IceBox server's configuration properties. For example, consider the following server configuration:

```
IceBox.Service.Weather=... --Ice.Config=svc.cfg
Ice.Trace.Network=1
```

The `Weather` service only receives the properties that are defined in its `IceBox.Service` property. In the example above, the service's communicator is initialized with the properties from the file `svc.cfg`.

If services need to inherit the IceBox server's configuration properties, define the `IceBox.InheritProperties` property in the IceBox server's configuration:

```
IceBox.Service.Weather=... --Ice.Config=svc.cfg
Ice.Trace.Network=1
IceBox.InheritProperties=1
```

All services inherit the server's properties when `IceBox.InheritProperties` is set to a non-zero value. The service inherits all the properties of the IceBox servers, with the exception of properties whose names start with `Ice.Admin`.

> The properties of the shared communicator are also affected by this setting.

Logging Considerations for IceBox Services

The IceBox server only configures a logger for a service if that service has not already specified its own logger via the Ice.LogFile or Ice.UseSyslog properties.

See Also

- IceBox.*
- Developing IceBox Services
- IceStorm
- Collocated Invocation and Dispatch
- Miscellaneous Ice.* Properties

Starting the IceBox Server

Incorporating everything we discussed previously, we can now configure and start IceBox servers.

On this page:

Starting the C++ IceBox Server

The configuration file for our example C++ service is shown below:

```
IceBox.Service.Hello=HelloService:create
Hello.Endpoints=tcp -p 10001
```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can usually start the C++ IceBox server as follows:

C++11 C++98

```
icebox++11 --Ice.Config=config
```

```
icebox --Ice.Config=config
```

Additional command line options are supported, including those that allow the server to run as a Windows service or Unix daemon.

> **32-bit IceBox on 64-bit Linux**
> On 64-bit Linux, the 32-bit IceBox executables (when provided) are named `icebox32` (C++98) and `icebox32++11` (C++11).

Starting the C# IceBox Server

The configuration file for our example C# service is shown below:

```
IceBox.Service.Hello=helloservice.dll:HelloService
Hello.Endpoints=tcp -p 10001
```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can start the C# IceBox server as follows:

.NET Framework 4.5 .NET Core 2.0

```
iceboxnet --Ice.Config=config
```

```
dotnet iceboxnet.dll --Ice.Config=config
```

Starting the Java IceBox Server

Our Java configuration is nearly identical to the C++ version, except for the entry point specification:

```
IceBox.Service.Hello=HelloServiceI
Hello.Endpoints=tcp -p 10001
```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can start the Java IceBox server as follows:

Java Java Compat

```
java -jar icebox-3.7.0.jar --Ice.Config=config
```

```
java -jar icebox-compat-3.7.0.jar --Ice.Config=config
```

IceBox Server Failures

At startup, an IceBox server inspects its configuration for all properties having the prefix `IceBox.Service` and initializes each service. If initialization fails for a service, the IceBox server invokes the `stop` operation on any initialized services, reports an error, and terminates.

See Also

- Service Helper Class
- IceBox.*

IceBox Administration

An IceBox server internally creates an object called the service manager that is responsible for loading and initializing the configured services. You can optionally expose this object to remote clients, such as the IceBox and IceGrid administrative utilities, so that they can execute certain administrative tasks.

On this page:

IceBox Administrative Slice Interfaces

The Slice definitions shown below comprise the IceBox administrative interface:

Slice

```
module IceBox
{
    exception AlreadyStartedException {}
    exception AlreadyStoppedException {}
    exception NoSuchServiceException {}

    interface ServiceObserver
    {
        void servicesStarted(Ice::StringSeq services);
        void servicesStopped(Ice::StringSeq services);
    }

    interface ServiceManager
    {
        idempotent Ice::SliceChecksumDict getSliceChecksums();
        void startService(string service)
            throws AlreadyStartedException, NoSuchServiceException;
        void stopService(string service)
            throws AlreadyStoppedException, NoSuchServiceException;
        void addObserver(ServiceObserver* observer)
        void shutdown();
    }
}
```

The IceBox `ServiceManager` Interface

The `ServiceManager` interface provides access to the service manager object of an IceBox server. It defines the following operations:

- `getSliceChecksums`
  Returns a dictionary of checksums that allows a client to verify that it is using the same Slice definitions as the server.

- `startService`
  Starts a pre-configured service that is currently inactive. This operation cannot be used to add new services at run time, nor will it cause an inactive service's implementation to be reloaded. If no matching service is found, the operation raises `NoSuchServiceException`. If the service is already active, the operation raises `AlreadyStartedException`.
- `stopService`
  Stops an active service but does not unload its implementation. The operation raises `NoSuchServiceException` if no matching service is found, and `AlreadyStoppedException` if the service is stopped at the time `stopService` is invoked.
- `addObserver`
  Adds an observer that is called when IceBox services are started or stopped. The service manager ignores operations that supply a null proxy, or a proxy that has already been registered.
- `shutdown`
  Terminates the services and shuts down the IceBox server.

The IceBox `ServiceObserver` Interface

An administrative client that is interested in receiving callbacks when IceBox services are started or stopped must implement the `ServiceObserver` interface and register the callback object's proxy with the service manager using its `addObserver` operation. The `ServiceObserver` interface defines two operations:

- `servicesStarted`
  Invoked immediately upon registration to supply the current list of active services, and thereafter each time a service is started.
- `servicesStopped`
  Invoked whenever a service is stopped, and when the IceBox server is shutting down.

The IceBox server unregisters an observer if the invocation of either operation causes an exception.

Our discussion of IceGrid includes an example that demonstrates how to register a `ServiceObserver` callback with an IceBox server deployed with IceGrid.

Enabling the Service Manager

IceBox's administrative functionality is disabled by default. You can enable it using the Ice administrative facility by defining endpoints for the `Ice.Admin` object adapter with the property `Ice.Admin.Endpoints`.

> The `Ice.Admin` object adapter is enabled automatically in an IceBox server that is deployed by IceGrid.

With the administrative facility enabled, IceBox registers an administrative facet with the name `IceBox.ServiceManager`. We discuss the identity of the `admin` object below.

> Exposing the service manager makes an IceBox server vulnerable to denial-of-service attacks from malicious clients. Consequently, you should choose the endpoints and transports carefully.

IceBox Admin Facets

When you enable the service manager, IceBox adds it as a facet of the server's `admin` object. As a result, the identity of the service manager is the same as that of the `admin` object, and the name of its facet is `IceBox.ServiceManager`.

The identity of the `admin` object uses either a UUID or a statically-configured value for its category, and the value `admin` for its name. For example, consider the following property definitions:

```
Ice.Admin.Endpoints=tcp -h 127.0.0.1 -p 10001
Ice.Admin.InstanceName=IceBox
```

In this case, the identity of the `admin` object is `IceBox/admin`.

IceBox also creates in each service communicator (shared communicator and per-service communicator) all the built-in facets enabled on its main communicator, and adds all these facets, except the `Process` facet, to its admin object. These facets are named `IceBox.Service.`*`service-name`*`.`*`facet-name`*, where *service-name* corresponds to the service name (for example `Hello` or `IceStorm`), and *facet-name* is the name of the built-in facet (for example `Properties` or `Logger`).

> You can instruct IceBox to skip the admin facets for a specific service by setting the property `Ice.Admin.Enabled` to a numeric value (typically 0) in the configuration for that service.

IceBox Administrative Client Configuration

A client requiring administrative access to the service manager must first obtain (or be able to construct) a proxy for the `admin` object. The default identity of the `admin` object uses a UUID for its category, which means the client cannot predict the identity and therefore will be unable to construct the proxy itself. If the IceBox server is deployed with IceGrid, the client can use the technique described in our discussion of IceGrid to access its `admin` object.

In the absence of IceGrid, the IceBox server should set the `Ice.Admin.InstanceName` property if remote administration is required. In so doing, the identity of the `admin` object becomes well-known, and a client can construct the proxy on its own. For example, let's assume that the IceBox server defines the following property:

```
Ice.Admin.InstanceName=IceBox
```

A client can define the proxy for the `admin` object in a configuration property as follows:

```
ServiceManager.Proxy=IceBox/admin -f IceBox.ServiceManager -h 127.0.0.1
-p 10001
```

The proxy option `-f IceBox.ServiceManager` specifies the name of the service manager's administrative facet.

IceBox Administrative Utility

IceBox includes C++ and Java implementations of an administrative utility. The utilities have the same usage:

```
Usage: iceboxadmin [options] [command...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.

Commands:
start SERVICE        Start a service.
stop SERVICE         Stop a service.
shutdown             Shutdown the server.
```

The C++ utility is named `iceboxadmin`. The Java utility is represented by the class `com.zeroc.IceBox.Admin` and the Java Compat

utility is `IceBox.Admin`.

The `start` command is equivalent to invoking `startService` on the service manager interface. Its purpose is to start a pre-configured service; it cannot be used to add new services at run time. Note that this command does not cause the service's implementation to be reloaded.

Similarly, the `stop` command stops the requested service but does not cause the IceBox server to unload the service's implementation.

The `shutdown` command stops all active services and shuts down the IceBox server.

The C++ and Java utilities obtain the service manager's proxy from the property `IceBoxAdmin.ServiceManager.Proxy`, therefore this proxy must be defined in the program's configuration file or on the command line, and the proxy's contents of depend on the server's configuration. If the IceBox server is deployed with IceGrid, we recommend using the IceGrid administrative utilities instead, which provide equivalent commands for administering an IceBox server. Otherwise, the proxy should have the endpoints and identity configured for the server.

See Also

- Slice Checksums
- Administrative Facility
- The admin Object
- The Properties Facet
- icegridadmin Command Line Tool
- IceGrid and the Administrative Facility
- IceBox.*
- IceBoxAdmin
- Ice.Admin.*

Ice Plugins

This chapter describes all the Ice plug-ins included with Ice. These plug-ins rely on the Plug-In Facility described earlier.

Topics

IceIAP

IceIAP is a transport plug-in that enables clients to communicate via the Apple iAP protocol reserved for accessories.

On this page:

IceIAP Overview

IceIAP is an Ice plug-in that must be installed in your iOS client applications that need to communicate with accessories over Bluetooth, the Apple Lightning connector, or the Apple 30-pin connector. This section reviews some concepts that will help you as you learn more about IceIAP.

The IceIAP transport is based on Apple's External Accessory framework and enables Ice clients running on iOS devices to communicate with Ice servers running on connected accessories. This transport is a client-side only transport for iOS. It doesn't for instance provide the server-side transport that is required on the accessory side. For information on how to implement the server side, you need to be a MFI licensee and get in touch with ZeroC.

Accessory Discovery

An accessory can be discovered based on a number of attributes:

- its name
- its manufacturer
- its model number
- an advertised protocol

An accessory endpoint can be configured with any of these attributes to find an accessory.

Installing IceIAP

The IceIAP plug-in must be installed in every client that needs to communicates with accessories. The plug-in is only distributed as a static library named `IceIAP`. Your C++ or Objective-C clients should link with this library and register the plug-in with one of the following functions:

C++ ObjC

```
Ice::registerIceIAP(bool loadOnInitialize = true)
```

```
ICEregisterIceIAP(BOOL loadOnInitialize)
```

This function must be called before communicator initialization. The `loadOnInitialize` parameter specifies if the plug-in is installed when the communicator is initialized. If set to `false`, you will need to enable the plug-in by setting the `Ice.Plugin.IceIAP` property to `1`.

> `Ice::registerIceIAP` is a simple helper function that calls `Ice::registerPluginFactory`.

Refer to the next section for information on configuring the plug-in.

Using IceIAP

This section describes how to incorporate IceIAP into your Ice applications.

Proxy Endpoints

An iAP endpoint in a proxy specifies attributes that are used to find and connect to a matching accessory. An iAP endpoint has the following syntax:

```
iap [-p PROTOCOL] [-n NAME] [-m MANUFACTURER] [-o MODELNUMBER]
```

For example, to invoke on a proxy for the `hello` object running on an accessory that implements the `com.zeroc.helloWorld` protocol, use the following stringified proxy:

```
hello:iap -p com.zeroc.helloWorld
```

To use the secured iAP endpoint, replace `iap` with `iaps`. You will also need to ensure that the IceSSL plug-in is loaded and configured when using `iaps` endpoints.

See Also

- Plug-in Facility
- IceSSL
- Proxy and Endpoint Syntax

IceBT

IceBT is a transport plug-in that enables clients and servers to communicate via Bluetooth RFCOMM connections on Android and Linux platforms.

On this page:

IceBT Overview

IceBT is an Ice plug-in that must be installed in all of the clients and servers in your application that need to communicate over Bluetooth. This section reviews some concepts that will help you as you learn more about IceBT.

Service Discovery

The Bluetooth specification defines a standard mechanism for discovering services called the Service Discovery Protocol (SDP). It's a flexible but complex specification that accommodates a wide range of Bluetooth device functionality and requirements. Fortunately, Ice users only need a passing familiarity with SDP.

The operating system's Bluetooth stack implements an SDP service that provides two basic functions: query and registration. IceBT considers each object adapter endpoint in an Ice server to be a service and adds a corresponding entry for it in the local SDP registry. This entry associates a UUID with a human-friendly name and an RFCOMM channel. For example, an entry might contain:

SDP Entry

```
Name: My Bluetooth Service
UUID: 1c6a142a-aae6-4d58-bef8-33196f531da7
RFCOMM: Channel #8
```

The SDP entry automatically expires when its service terminates.

An Ice client requires two values to connect to a server:

1. The server's Bluetooth device address (such as `01:23:45:67:89:AB`)
2. The UUID of the desired service

To establish a connection, the client first queries the SDP service on the server device for an entry matching the target UUID. If a match is found, the SDP service returns the server's current RFCOMM channel, and the client open a connection to that channel.

IceBT takes care of all of this for you during server initialization and connection establishment.

Device Discovery

When developing a client application, you'll normally hard-code the UUIDs of the remote services that your client requires because those UUIDs must match the ones advertised by your servers. However, in addition to a UUID, a client also needs to know the device address on which a service is running. Typically the client will use the system's Bluetooth API to initiate device discovery and present the results to the user. We discuss this further in the "Using IceBT" section below.

Installing IceBT

The IceBT plug-in must be installed in every client and server that needs to communicate via Bluetooth.

You can use the `Ice.Plugin` property to load and install the plug-in at run time; the property value depends on the language mapping you're using:

C++ Java Java Compat Python Ruby PHP

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

```
# Android only
Ice.Plugin.IceBT=com.zeroc.IceBT.PluginFactory
```

```
# Android only
Ice.Plugin.IceBT=IceBT.PluginFactory
```

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

```
# Linux only
Ice.Plugin.IceBT=IceBT:createIceBT
```

If using C++, instead of dynamically loading the plug-in at run time your application can explicitly link with and register the plug-in. To register the plug-in, you must call the `Ice::registerIceBT(bool loadOnInitialize = true)` function before the communicator initialization. The `loadOnInitialize` parameter specifies if the plug-in is installed when the communicator is initialized. If set to `false`, you will need to enable the plug-in by setting the `Ice.Plugin.IceBT` property to `1`.

> `Ice::registerIceBT` is a simple helper function that calls `Ice::registerPluginFactory`.

Refer to the next section for information on configuring the plug-in.

Configuring IceBT

The IceBT plug-in supports some new configuration properties, including settings to modify the size of the send and receive buffers for a connection. The default settings should be sufficient for most applications.

Developers should also be aware of some core Ice properties that can affect Bluetooth connections:

- Default device address – If you omit a device address from an object adapter endpoint or proxy endpoint, the plug-in defaults to the address specified by the property `Ice.Default.Host`.
- Connect timeout – Establishing a Bluetooth connection can take several seconds to complete. Ice's default timeout settings give

plenty of time for a connection to succeed, but an application could experience problems if it configures custom connection timeouts that are too small for Bluetooth connections.

Using IceBT

This section describes how to incorporate IceBT into your Ice applications.

Object Adapter Endpoints

A Bluetooth "service" corresponds to an Ice endpoint, and each endpoint requires its own UUID.

> On Linux, you can use the `uuidgen` command to generate new UUIDs. Web-based UUID generators are also available.

For example, using the syntax for Bluetooth endpoints, you can configure an object adapter named `HelloService` as follows:

```
HelloService.Endpoints=bt -u 4f140cef-d75e-4c93-b4e4-20ac111d36d1 --name
"Hello Service"
```

We're associating the UUID `4f140cef-d75e-4c93-b4e4-20ac111d36d1` with our service. At run time, this service will be advertised in the Service Discovery Protocol (SDP) registry along with the descriptive name `Hello Service`. We omitted a device address, so the plug-in will listen on the host's default Bluetooth adapter. We also did not specify a particular RFCOMM channel (using the `-c` option) and therefore the plug-in will automatically select an available channel.

If you omit the `-u UUID` option from the object adapter's endpoint, the plug-in will automatically generate a random UUID for use in the SDP registry. Note however that your clients will still need some way of discovering this UUID. Generally speaking, you should generate and use your own well-known UUIDs instead.

> On Linux, use the `sdptool` command to view the contents of the SDP registry on a device:
>
> ```
> > sdptool browse local
> > sdptool browse 01:23:45:67:89:AB
> ```
>
> The first command displays the active services of the local host, and the second command shows the active services of a remote device.

Proxy Endpoints

A Bluetooth endpoint in a proxy must include a UUID and a device address:

C++11 C++98

```
auto proxy = communicator->stringToProxy("hello:bt -u
4f140cef-d75e-4c93-b4e4-20ac111d36d1 -a \"01:23:45:67:89:AB\"");
```

```
Ice::ObjectPrx proxy = communicator->stringToProxy("hello:bt -u
4f140cef-d75e-4c93-b4e4-20ac111d36d1 -a \"01:23:45:67:89:AB\"");
```

The UUID specified with the `-u` option must match the one you assigned to your object adapter endpoint.

Notice that the device address given by the `-a` option is enclosed in quotes; this is necessary because colon (`:`) characters are used as separators in stringified proxies.

You can omit a device address if you define `Ice.Default.Host`.

Refer to Proxy and Endpoint Syntax for complete details on the format of a Bluetooth endpoint.

Applications are responsible for determining the Bluetooth address of the device hosting the target service, as described in the next section.

Implementing Discovery

Device discovery is a platform-specific activity that applications are responsible for implementing. On Android, an app can use the APIs in an `droid.bluetooth` to initiate discovery and receive intent notifications about nearby devices. The Android sample program `talk` (see below) shows how to implement discovery.

On Linux, the IceBT plug-in provides a C++ API for device discovery:

C++11 C++98

```
namespace IceBT
{
    using PropertyMap = std::map<std::string, std::string>;

    class Plugin : public Ice::Plugin
    {
    public:

         virtual void startDiscovery(const std::string& address,
                                     std::function<void(const
std::string& addr, const PropertyMap& props)>) = 0;
         virtual void stopDiscovery(const std::string& address) = 0;
    };
}
```

```
namespace IceBT
{
    typedef std::map<std::string, std::string> PropertyMap;

    class DiscoveryCallback : public IceUtil::Shared
    {
    public:
        virtual void discovered(const std::string& address, const
PropertyMap& props) = 0;
    };
    typedef IceUtil::Handle<DiscoveryCallback> DiscoveryCallbackPtr;

    class Plugin : public Ice::Plugin
    {
    public:

        virtual void startDiscovery(const std::string& address, const
DiscoveryCallbackPtr& cb) = 0;
        virtual void stopDiscovery(const std::string& address) = 0;
    };
    typedef IceUtil::Handle<Plugin> PluginPtr;
}
```

An application must implement a callback function or object and pass it to `startDiscovery`:

C++11 C++98

```
#include <IceBT/IceBT.h>
...

auto communicator = ...
auto plugin = communicator->getPluginManager()->getPlugin("IceBT");
auto btplugin = dynamic_pointer_cast<IceBT::Plugin>(plugin);
btplugin->startDiscovery("", [](const std::string& addr, const
PropertyMap& props) { ... });
```

```
#include <IceBT/IceBT.h>
...
class DiscoveryCallbackI : public IceBT::DiscoveryCallback
{
public:

    virtual void discovered(const std::string& address, const
IceBT::PropertyMap& props)
    {
        ...
    }
};
...
Ice::CommunicatorPtr communicator = ...
Ice::PluginPtr plugin =
communicator->getPluginManager()->getPlugin("IceBT");
IceBT::PluginPtr btplugin = IceBT::PluginPtr::dynamicCast(plugin);
btplugin->startDiscovery("", new DiscoveryCallbackI);
```

For each nearby device discovered by the Bluetooth stack, the plug-in will invoke the callback. The arguments to the callback are the Bluetooth address of the nearby device and a string map of properties containing metadata about that device. As shown in the example above, the application can pass an empty string to `startDiscovery` and the plug-in will use the default Bluetooth adapter, otherwise the application can pass the device address of the desired adapter.

Discovery will continue until `stopDiscovery` is called or a Bluetooth connection is initiated.

Connection Limitations

Be aware of the following limitations when using IceBT:

- An application cannot open multiple Bluetooth connections to the same remote endpoint. This is not a limitation in Ice but rather in the Bluetooth stack. Normally this limitation won't impact your application because Ice's default behavior is to reuse an existing connection to an endpoint whenever possible in preference to opening a new connection. However, some application designs may attach additional semantics to a connection, using Ice APIs to override the default behavior and force the establishment of new connections to the same endpoint. This strategy will not work when using Bluetooth.
- If a Linux server forcefully closes a Bluetooth connection, the connection loss may not be detected by the client. It's important to configure sensible connection timeouts and invocation timeouts to avoid lengthy delays.

IceBT Sample Programs

The ice-demos repository includes a C++ command-line program for Linux (in `cpp/IceBT/talk`) and an app for Android (in `java/Android/talk`). These programs allow two devices to talk to one another via Bluetooth.

Security Notes for IceBT

The Bluetooth stack performs its own encryption of transmitted data using keys generated during the pairing process. Two devices must already be paired before Ice applications on those devices can communicate with one another. IceBT does not implement or provide an API for pairing; rather, this is something that is normally done at the user level. However, it's possible that an Ice connection attempt will *initiate* a pairing process that the user must then complete.

Android provides two APIs for establishing a connection: a secure version and an insecure version. The difference between the two lies in the pairing behavior, where the secure version causes the system to prompt the user (if necessary) and the insecure version does not. IceBT always uses the secure API.

For added security, you can use SSL over Bluetooth by installing the IceSSL plug-in. During its initialization, the IceBT plug-in checks for the

presence of IceSSL and, if it's found, IceBT adds a second transport protocol named `bts`. No additional changes are necessary to your endpoints, and you can use the IceSSL configuration properties as usual to define your security settings.

See Also

- IceBT.*
- Plug-in Facility
- Proxy and Endpoint Syntax

IceDiscovery

IceDiscovery provides a location service using UDP multicast that enables Ice applications to discover objects and object adapters.

On this page:

IceDiscovery Overview

IceDiscovery is an Ice plug-in that must be installed in all of the clients and servers in your application. Once installed, IceDiscovery enables a client to dynamically locate objects using indirect proxies, which avoids the need for the client to statically configure the endpoints of the objects it uses. In a server, IceDiscovery makes objects and object adapters available for discovery with minimal additional effort.

IceDiscovery Concepts

This section reviews some concepts that will help you as you learn more about IceDiscovery.

Indirect Proxies

Indirect proxies have two formats:

- `identityOnly`
  This format (well-known proxy) uses only the object's identity.
- `identity@adapterId`
  This format combines the object identity and an adapter identifier. This identifier can either refer to a specific object adapter or a replica group.

Notice that neither format includes any endpoints, such as `tcp -h somehost -p 10000`. The ability to resolve an indirect proxy using only symbolic information, much like a DNS lookup, helps to loosen the coupling between clients and servers.

The Ice core delegates the resolution of indirect proxies to a standardized locator facility. This architecture offers a significant advantage: Ice applications can change locator settings via external configuration without requiring any changes to the application code.

Plug-in Facility

In addition to the locator facility, Ice also defines a standard plug-in facility through which applications can modify the functionality of the Ice core or add new capabilities, again using only external configuration.

IceDiscovery combines these two facilities: it's installed as a standard Ice plug-in via configuration, and this plug-in installs a custom locator implementation that enables discovery using UDP multicast.

We'll describe later how to install IceDiscovery in your clients and servers.

Replication

The locator facility includes support for replicated object adapters. For example, if `Adapter1` and `Adapter2` both participate in a replicate group identified as `TheGroup`, then the indirect proxy `someObject@TheGroup` could resolve to either `someObject@Adapter1` or `someOb`

`ject@Adapter2`. Although there are two adapters that host `someObject` at the implementation level, both object adapters incarnate the same logical object. The application is responsible for ensuring that any persistent state is properly synchronized between the servers that host replicated object adapters.

IceDiscovery Domains

Nothing prevents two unrelated IceDiscovery applications from using the same multicast address and port, which means a plug-in from Application A can receive lookup requests from a client in Application B for objects and object adapters that might coincidentally match those of Application A. To avoid this situation, the applications should configure unique *domain identifiers*. The client plug-in includes its domain identifier in each lookup request it sends so that a server plug-in can ignore any requests that don't match its own domain identifier.

Discovery Process

When a client uses an indirect proxy with an adapter ID for the first time:

- The Ice run time queries the Ice locator implemented by the IceDiscovery plug-in to resolve the indirect proxy's adapter ID.
- The client's IceDiscovery plug-in transparently broadcasts a `findAdapterById` request via multicast.
- The lookup request includes the adapter ID, and a proxy that the target server's IceDiscovery plug-in can use to communicate directly with the client's IceDiscovery plug-in.
- Every server that installs the plug-in and uses the same addressing information receives the client's multicast lookup request; only the server that hosts the target adapter sends a reply.
- The client plug-in waits for a reply using a configurable timeout period; if it doesn't receive a reply, it tries again a configurable number of times before giving up.
- The target server's IceDiscovery plug-in sends a reply including a template proxy for the target adapter and whether or not it's replicated.
- The client's plug-in receives the reply and:
  - if the adapter is not replicated, it returns the proxy to the Ice run time location facility.
  - if the adapter is replicated, it waits again for replies from other servers. The duration of the wait is based on the time it took to receive the first reply (the latency) and a configurable latency multiplier.
- The Ice run time location facility caches the endpoints for the object adapter and the client's application code uses these cached endpoints to communicate directly with the target object using whichever transports its object adapter supports.

When the client uses a well-known proxy for the first time, an additional step occurs:

- The Ice run time queries the Ice locator implemented by the IceDiscovery plug-in to resolve the well-known proxy.
- The client's IceDiscovery plug-in transparently broadcasts a `findObjectById` request via multicast.
- The lookup request includes the identity of the target object, and a proxy that the target server's IceDiscovery plug-in can use to communicate directly with the client's IceDiscovery plug-in.
- Every server that installs the plug-in and uses the same addressing information receives the client's multicast lookup request; only the server that hosts this object sends a reply.
  - To check if a server hosts this object, the IceDiscovery plugin in the server searches the object adapters that have registered with the IceDiscovery LocatorRegistry by attempting to ping the object in these object adapters with `ice_ping`. It firsts checks the object adapters registered with a replica group ID, and then the object adapters registered without a replica group ID. This way, the object may be incarnated by a servant in the Active Servant Map, or by a default servant, or by a servant returned by a servant locator.
- The client plug-in waits for a reply using a configurable timeout period; if it doesn't receive a reply, it tries again a configurable number of times before giving up.
- The target server's IceDiscovery plug-in sends a reply including an indirect proxy for the target object.
- The Ice runtime location facility caches the indirect proxy for the well-known object.
- The indirect proxy is resolved with IceDiscovery using the steps mentioned above for indirect proxies.

Unless the Ice locator cache is disabled, only the initial lookup request occurs over multicast. Further requests use the information from the Ice runtime locator cache. The reply from the server plug-in to the client plug-in occurs using UDP unicast (by default). All subsequent communication between the client and the target object proceed directly without intervention by the IceDiscovery plug-in.

IceDiscovery vs. IceGrid

IceDiscovery and IceGrid both provide a location service but it helps to understand their differences when deciding which one to use in an application. Use IceDiscovery when your application needs a lightweight, transient location service. IceGrid's location service is backed by a persistent database and represents just one of the features that IceGrid offers, along with remote administration, on-demand server activation, and many others. If you need a location service but aren't yet ready to dive into IceGrid, start out using IceDiscovery; migrating to IceGrid later won't be difficult.

> We provide a plug-in similar to IceDiscovery called IceLocatorDiscovery that integrates with IceGrid.

Installing IceDiscovery

The IceDiscovery plug-in must be installed in every client that needs to locate objects and in all of the servers that host those objects.

You can use the `Ice.Plugin` property to load and install the plug-in at runtime; the property value depends on the language mapping you're using:

C++ Java Java Compat C# Python Ruby PHP

```
Ice.Plugin.IceDiscovery=IceDiscovery:createIceDiscovery
```

```
Ice.Plugin.IceDiscovery=IceDiscovery:com.zeroc.IceDiscovery.PluginFacto
ry
```

```
Ice.Plugin.IceDiscovery=IceDiscovery:IceDiscovery.PluginFactory
```

```
Ice.Plugin.IceDiscovery=IceDiscovery:IceDiscovery.PluginFactory
```

```
# Uses the C++ plug-in
Ice.Plugin.IceDiscovery=IceDiscovery:createIceDiscovery
```

```
# Uses the C++ plug-in
Ice.Plugin.IceDiscovery=IceDiscovery:createIceDiscovery
```

```
# Uses the C++ plug-in
Ice.Plugin.IceDiscovery=IceDiscovery:createIceDiscovery
```

The C++ configuration is the same for the C++11 mapping and the C++98 mapping: Ice computes the name of the shared library to load and adds automatically a "++11" suffix when needed.

If using C++, instead of dynamically loading the plug-in at run time your application can explicitly link with and register the plug-in. To register the plug-in, you must call the `Ice::registerIceDiscovery(bool loadOnInitialize = true)` function before the communicator initialization. The `loadOnInitialize` parameter specifies if the plug-in is installed when the communicator is initialized. If set to `false`, you will need to enable the plugin by setting the `Ice.Plugin.IceDiscovery` property to `1`.

> `Ice::registerIceDiscovery` is a simple helper function that calls `Ice::registerPluginFactory`.

Refer to the next section for information on configuring the plug-in.

Configuring IceDiscovery

Applications configure the IceDiscovery plug-in using configuration properties; the plug-in does not provide a local API.

IceDiscovery Property Overview

The IceDiscovery plug-in supports a number of configuration properties, most of which affect the endpoints that the plug-in uses to communicate with its peers:

- Lookup endpoint
  This is the multicast endpoint on which all lookup queries are broadcast. It must use an IPv4 or IPv6 address in the multicast range with a fixed port.
- Reply endpoint
  This is the endpoint on which a client receives replies from servers. In general, this endpoint should not use a fixed port.

The plug-in uses sensible default values for all of its configuration properties, such that it's often unnecessary to define any of the plug-in's properties. However, it's still important to understand how the plug-in derives its endpoint information.

IceDiscovery creates several object adapters in each communicator in which it's installed, including the object adapters `IceDiscovery.Multicast` and `IceDiscovery.Reply`. These object adapters correspond to the Lookup and Reply endpoints mentioned above, respectively. You can configure the endpoints of these object adapters directly by defining the properties `IceDiscovery.Multicast.Endpoints` and `IceDiscovery.Reply.Endpoints`. If you don't define an endpoint for an object adapter, the plug-in computes it as follows:

- `IceDiscovery.Multicast.Endpoints=udp -h` *`address`* `-p` *`port`* `[--interface` *`interface`*`]`
- `IceDiscovery.Reply.Endpoints=udp [--interface` *`interface`*`]`

where

- *`address`* is the value of `IceDiscovery.Address` - defaults to `239.255.0.1` if IPv4 is enabled or `ff15::1` if IPv4 is disabled
- *`port`* is the value of `IceDiscovery.Port` - defaults to `4061`
- *`interface`* is the value of `IceDiscovery.Interface`

Consequently, if you don't define any of these properties, the plug-in uses the following endpoints by default (assuming IPv4):

- `IceDiscovery.Multicast.Endpoints=udp -h 239.255.0.1 -p 4061`
- `IceDiscovery.Reply.Endpoints=udp`

Finally, you can also override the default endpoint that a client uses to broadcast its lookup queries by defining `IceDiscovery.Lookup`, otherwise the plug-in computes this endpoint as follows:

- `IceDiscovery.Lookup=udp -h` *`address`* `-p` *`port`* `[--interface` *`interface`*`]`

This endpoint must use the same address and port as `IceDiscovery.Multicast.Endpoints`.

As you can see, the properties `IceDiscovery.Address`, `IceDiscovery.Port` and `IceDiscovery.Interface` are simply used as convenient shortcuts for customizing the details of the plug-in's endpoints. For example, suppose we want to use a different multicast address and port:

```
IceDiscovery.Address=239.255.0.99
IceDiscovery.Port=8000
```

The plug-in derives the following properties from these settings:

```
IceDiscovery.Multicast.Endpoints=udp -h 239.255.0.99 -p 8000
IceDiscovery.Lookup=udp -h 239.255.0.99 -p 8000
```

> All of the clients and servers comprising an application must use the same values for `IceDiscovery.Address` and `IceDiscovery.Port`. You should also consider defining `IceDiscovery.DomainId` to avoid any potential collisions from unrelated applications that happen to use the same address and port.

Configuring IceDiscovery in Clients

Aside from installing the plug-in and optionally configuring its addressing information, no other configuration steps are required for an IceDiscovery client.

Configuring IceDiscovery in Servers

In addition to installing the plug-in and optionally configuring its addressing information, you also need to configure an identifier for each of a server's object adapters that hosts well-known (discoverable) objects. For example, suppose a server creates an object adapter named `Hello` and we want its objects to be discoverable. We can configure the object adapter's `AdapterId` property as follows:

```
Hello.AdapterId=HelloAdapter
```

The identifier you select must be globally unique among the servers sharing the same address and domain settings.

To use object adapter replication, you'll need to include the `ReplicaGroupId` property for each replicated object adapter:

```
Hello.AdapterId=Hello1
Hello.ReplicaGroupId=HelloAdapter
```

The indirect proxy `someObject@Hello1` refers to an object in this particular object adapter, whereas the indirect proxy `someObject@HelloAdapter` could refer to any object having the identity `someObject` in any of the object adapters participating in the replica group `HelloAdapter`.

Configuring a Locator Proxy

The IceDiscovery plug-in calls `setDefaultLocator` on its communicator at startup, therefore it's not necessary for you to configure a locator proxy.

Using IceDiscovery

As its name implies, the IceDiscovery plug-in enables clients to locate objects at run time, as long as the servers hosting those objects are actively running and using the same configuration settings for address, port, domain, and so on. Since IceDiscovery relies on UDP multicast to broadcast the lookup requests, you'll need to ensure that your network supports this transport.

IceDiscovery Design Decisions

From a design perspective, incorporating IceDiscovery into your application requires answering the following questions:

- What is the set of well-known objects that will be available for discovery?
  Applications typically don't need to make *every* object available for discovery. Rather, designs often only make "bootstrap" or "factory" objects available for discovery, while proxies for other objects can be obtained by invoking operations on these initial objects.
- How should clients refer to these well-known objects?
  As we explained earlier, proxies for well-known objects can take two forms: an identity by itself, or an identity with an adapter identifier, such as `factory` and `factory@AccountAdapter`, respectively. Clearly, using only identities places a greater burden on the application to ensure that they are globally unique among all of the clients and servers sharing the same address and domain settings. Including an object adapter identifier can help to further partition the object namespace, so that `factory@AccountAdapter` and `factory@AdminAdapter` represent distinct objects without requiring artificially unique identities such as `accountFactory` and `adminFactory`.
- Can unrelated applications share the same multicast address and port?
  If so, select unique domain identifiers for the applications and configure `IceDiscovery.DomainId` properties to avoid the

potential for subtle bugs.

- Do you need replicated objects?
  Replicated object adapters can improve the fault tolerance of your application by allowing independent server processes to implement the same logical objects. Configure your servers as described above, and ensure your clients resolve indirect proxies using the replica group identifiers.

IceDiscovery provides enough flexibility to support a wide variety of application architectures. If you need additional functionality, consider using IceGrid instead. Note also that IceGrid supports its own version of IceDiscovery, so that migrating an existing IceDiscovery application should be straightforward.

IceDiscovery Sample Programs

The Ice distribution includes two IceDiscovery sample programs:

- `hello` - A basic client/server application
- `replication` - An application that demonstrates the use of object adapter replication

Refer to the `README` files in the demo source directories for more information on these examples.

See Also

- IceDiscovery.*
- Plug-in Facility
- Locators
- Well-Known Objects
- IceGrid

IceLocatorDiscovery

This page describes IceLocatorDiscovery, an Ice plug-in that enables the discovery of IceGrid and custom locators via UDP multicast.

On this page:

IceLocatorDiscovery Overview

IceLocatorDiscovery is an Ice plug-in that discovers IceGrid and custom locators on a network using UDP multicast. Once installed, the plug-in automatically and transparently issues a multicast query in an attempt to find one or more locators, collects the responses, and configures the Ice run time accordingly. The primary advantage of using IceLocatorDiscovery is that it eliminates the need to manually configure and maintain the `Ice.Default.Locator` property. It's even more helpful in a replicated IceGrid deployment consisting of a master replica and one or more slave replicas, where the `Ice.Default.Locator` property would normally include endpoints for some or all of the replicas. Avoiding the need to statically configure the locator endpoints relieves some of the administrative burden, simplifies deployment and configuration tasks, and adds more flexibility to your application designs.

> You can think of IceLocatorDiscovery as an application-specific version of IceDiscovery geared primarily toward IceGrid users.

Installing IceLocatorDiscovery

The IceLocatorDiscovery plug-in must be installed in every client that needs to locate objects; you can optionally install it in IceGrid nodes and registry replicas.

You can use the `Ice.Plugin` property to install the plug-in; the property value depends on the language mapping you're using:

C++ Java Java Compat C# Python Ruby PHP

```
Ice.Plugin.IceLocatorDiscovery=IceLocatorDiscovery:createIceLocatorDisc
overy
```

```
Ice.Plugin.IceLocatorDiscovery=IceLocatorDiscovery:com.zeroc.IceLocator
Discovery.PluginFactory
```

```
Ice.Plugin.IceLocatorDiscovery=IceLocatorDiscovery:IceLocatorDiscovery.
PluginFactory
```

```
Ice.Plugin.IceLocatorDiscovery=IceLocatorDiscovery:IceLocatorDiscovery.
PluginFactory
```

```
# Uses the C++ plug-in
Ice.Plugin.IceLocatorDiscovery=IceLocatorDiscovery:createIceLocatorDisc
overy
```

```
# Uses the C++ plug-in
Ice.Plugin.IceLocatorDiscovery=IceLocatorDiscovery:createIceLocatorDisc
overy
```

```
# Uses the C++ plug-in
Ice.Plugin.IceLocatorDiscovery=IceLocatorDiscovery:createIceLocatorDisc
overy
```

The C++ configuration is the same for the C++11 mapping and the C++98 mapping: Ice computes the name of the shared library to load and adds automatically a "++11" suffix when needed.

If using C++, instead of dynamically loading the plug-in at run time, your application can explicitly link with and register the plug-in. To register the plug-in, you must call the `Ice::registerIceLocatorDiscovery(bool loadOnInitialize = true)` function before the communicator initialization. The `loadOnInitialize` parameter specifies if the plug-in is installed when the communicator is initialized. If set to `false`, you will need to enable the plugin by setting the `Ice.Plugin.IceLocatorDiscovery` property to `1`.

> `Ice::registerIceLocatorDiscovery` is a simple helper function that calls `Ice::registerPluginFactory`.

Refer to the next section for information on configuring the plug-in.

Configuring IceLocatorDiscovery

Applications configure the IceLocatorDiscovery plug-in using configuration properties; the plug-in does not provide a local API.

IceLocatorDiscovery Property Overview

The IceDiscovery plug-in supports a number of configuration properties, many of which affect the endpoints that the plug-in uses for its queries:

- Lookup endpoint
  This is the multicast endpoint on which all lookup queries are broadcast. It must use an IPv4 or IPv6 address in the multicast range with a fixed port.
- Reply endpoint
  This is the endpoint on which the plug-in receives replies from locators (an IceGrid registry is the most common example of a locator). In general, this endpoint should not use a fixed port.

The plug-in uses sensible default values for all of its configuration properties, such that it's often unnecessary to define any of the plug-in's properties. However, it's still important to understand how the plug-in derives its endpoint information.

First, you can override the default endpoint that the plug-in uses to broadcast its queries by defining `IceLocatorDiscovery.Lookup`, otherwise the plug-in computes this endpoint as follows:

```
IceLocatorDiscovery.Lookup=udp -h address -p port [--interface interface]
```

where

- *address* is the value of `IceLocatorDiscovery.Address` - defaults to `239.255.0.1` if IPv4 is enabled or `ff15::1` if IPv4 is disabled
- *port* is the value of `IceLocatorDiscovery.Port` - defaults to `4061`
- *interface* is the value of `IceLocatorDiscovery.Interface`

> For IceGrid users, the lookup endpoint must use the same multicast address and port as `IceGrid.Registry.Discovery.Endpoints` in the registry configuration.

IceLocatorDiscovery also creates object adapters in each communicator in which it's installed, including the object adapter `IceLocatorDiscovery.Reply`. This object adapter corresponds to the Reply endpoint mentioned above.

As you can see, the properties `IceLocatorDiscovery.Address`, `IceLocatorDiscovery.Port` and `IceLocatorDiscovery.Interface` are simply used as convenient shortcuts for customizing the details of the plug-in's endpoints. For example, suppose we want to use a different multicast address and port:

```
IceLocatorDiscovery.Address=239.255.0.99
IceLocatorDiscovery.Port=8000
```

The plug-in derives the following property from these settings:

```
IceLocatorDiscovery.Lookup=udp -h 239.255.0.99 -p 8000
```

> All of the components of an IceGrid application must use the same multicast address and port. You should also consider defining `IceLocatorDiscovery.InstanceName` to avoid any potential collisions from unrelated IceGrid applications that happen to use the same address and port.

Configuring IceLocatorDiscovery in User Applications

For a client application, remove any existing definition of `Ice.Default.Locator`, then install the plug-in and optionally configuring its addressing information.

For a server deployed with IceGrid, you normally don't need to install the IceLocatorDiscovery plug-in.

Configuring IceLocatorDiscovery in IceGrid Administrative Clients

Support for multicast discovery is built into the command-line and graphical IceGrid administrative utilities, therefore you don't need to install the plug-in. Both utilities support configuration properties similar to the ones we described above for defining the multicast address and port.

Configuring IceLocatorDiscovery in an IceGrid Registry

An IceGrid registry does not need the plug-in if it's the master replica or the only registry in a deployment, although there's no harm in installing it. The plug-in is useful for slave replicas because it allows them to locate the current master without statically configuring the master's endpoints. Configure the plug-in for slave replicas just like you would for any C++ client.

IceGrid registries listen for multicast discovery queries by default, but you can disable this feature by setting `IceGrid.Registry.Discovery.Enabled` to zero.

If you've changed the default multicast address or port for IceLocatorDiscovery, you must also make corresponding changes to the configuration of each registry. The registry supports properties similar to those of IceLocatorDiscovery:

- `IceGrid.Registry.Discovery.Address`
- `IceGrid.Registry.Discovery.Port`
- `IceGrid.Registry.Discovery.Interface`

These properties influence the endpoint on which the registry listens for multicast discovery queries. If you don't override the endpoint by setting `IceGrid.Registry.Discovery.Endpoints`, the registry uses these properties to compute its endpoint as follows:

```
IceGrid.Registry.Discovery.Endpoints=udp -h address -p port [--interface interface]
```

> You don't need to define any `IceGrid.Registry.Discovery.*` properties if you want the registry to listen for discovery queries on its default multicast address and port.

Configuring IceLocatorDiscovery in an IceGrid Node

The plug-in is useful for IceGrid nodes because it allows them to locate the registry without statically configuring the registry's endpoints. Configure nodes just like you would for any C++ client.

See Also

- IceLocatorDiscovery Properties
- Registry Replication
- Plug-in Facility

IceSSL

Security is an important consideration for many distributed applications, both within corporate intranets as well as over untrusted networks, such as the Internet. The ability to protect sensitive information, ensure its integrity, and verify the identities of the communicating parties is essential for developing secure applications. With those goals in mind, Ice includes the IceSSL *plug-in* that provides these capabilities using the Secure Socket Layer (SSL) protocol.

Although security is an optional component of Ice, it is not an afterthought. The IceSSL plug-in integrates easily into existing Ice applications, in most cases requiring nothing more than configuration changes. Naturally, some additional effort is required to create the necessary security infrastructure for an application, but in many enterprises this work will have already been done.

> IceSSL is available for C++, Java and .NET applications. Objective-C, Python, PHP and Ruby applications can use IceSSL for C++ via configuration.

On this page:

Overview of SSL

The Secure Socket Layer (SSL) protocol is the de facto standard for secure network communication. Its support for authentication, non-repudiation, data integrity, and strong encryption makes it the logical choice for securing Ice applications.

SSL is the protocol [1] that enables Web browsers to conduct secure transactions and therefore is one of the most commonly used protocols for secure network communication. You do not need to know the technical details of the SSL protocol in order to use IceSSL successfully (and those details are outside the scope of this text). However, it would be helpful to have a high-level understanding of how the protocol works and the infrastructure required to support it.

SSL provides a secure environment for communication (without sacrificing too much performance) by combining a number of cryptographic techniques:

- public key encryption
- symmetric (shared key) encryption
- message authentication codes
- digital certificates

When a client establishes an SSL connection to a server, a *handshake* is performed. During a typical handshake, digital certificates that identify the communicating parties are validated, and symmetric keys are exchanged for encrypting the session traffic. Public key encryption, which is too slow to be used for the bulk of a session's data transfer, is used heavily during the handshaking phase. Once the handshake is complete, SSL uses message authentication codes to ensure data integrity, allowing the client and server to communicate at will with reasonable assurance that their messages are secure.

Public Key Infrastructure

Security requires trust, and public key cryptography by itself does nothing to establish trust. SSL addresses the issue of trust using Public Key Infrastructure (PKI) [2], which binds public keys to identities using certificates. A certificate *issuer* creates a certificate for an entity, called the *subject*. The subject is often a person, but it may also be a computer or a specific application. The subject's identity is represented by a *distinguished name*, which includes information such as the subject's name, organization and location. A certificate alone is not sufficient to establish the subject's identity, however, as anyone can create a certificate for a particular distinguished name.

In order to authenticate a certificate, we need a third-party to guarantee that the certificate belongs to the subject described by the distinguished name. This third party, called a Certificate Authority (CA), expresses this guarantee by using its own private key to sign the subject's certificate. Combining the CA's certificate with the subject's certificate forms a *certificate chain* that provides SSL with most of the information it needs to authenticate the remote peer. In many cases, the chain contains only the aforementioned two certificates, but it is also possible for the chain to be longer when the *root* CA issues a certificate that the subject may use to sign other certificates. Regardless of the length of the chain, this scheme can only work if we trust that the root CA has sufficiently verified the identity of the subject before issuing the certificate.

An implementation of the SSL protocol also needs to know which root CAs we trust. An application supplies that information as a list of certificates representing the trusted root CAs. With that list in hand, the SSL implementation authenticates a peer by obtaining the peer's certificate chain and examining it carefully for validity. If we view the chain as a hierarchy with the root CA certificate at the top and the peer's certificate at the bottom, we can describe SSL's validation activities as follows:

- The root CA certificate must be self-signed and be present among the application's trusted CA certificates.
- All other certificates in the chain must be signed by the one immediately preceding it.
- The certificates must not be expired or revoked.

These tests certify that the chain is valid, but applications often require the chain to undergo more intensive scrutiny to determine whether the chain is trustworthy.

Commercial CAs exist to supply organizations with a reliable source of certificates, but in many cases a private CA is completely sufficient. You can create and manage your CA using freely-available tools, and in fact Ice includes a collection of utilities that simplify this process.

Depending on your implementation language, it may also possible to avoid the use of certificates altogether; encryption is still used to obscure the session traffic, but the benefits of authentication are sacrificed in favor of reduced complexity and administration.

Requirements

Integrating IceSSL into your application often requires no changes to your source code, but does involve the following administrative tasks:

- creating a public key infrastructure (if necessary)
- configuring the IceSSL plug-in
- modifying your application's configuration to install the IceSSL plug-in and use secure connections

The remainder of this discussion concentrates on plug-in configuration and programming.

Topics

References

1. Viega, J., et al. 2002. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly.
2. Housley, R., and T. Polk. 2001. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*. Hoboken, NJ: Wiley.

Using IceSSL

Incorporating IceSSL into your application requires installing the plug-in, configuring it according to your security requirements, and creating SSL endpoints.

On this page:

Installing IceSSL

Ice supports a generic plug-in facility that allows extensions (such as IceSSL) to be installed dynamically without changing the application source code. The `Ice.Plugin` property provides language-specific information that enables the Ice run time to install a plug-in.

C++ Applications

The executable code for the IceSSL C++ plug-in resides in a shared library on Unix and a dynamic link library (DLL) on Windows. The format for the `Ice.Plugin` property is shown below:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
```

The C++ configuration is the same for the C++11 mapping and the C++98 mapping: Ice computes the name of the shared library to load and adds automatically a "++11" suffix when needed.

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value `IceSSL:createIceSSL` is sufficient to allow the Ice run time to locate the IceSSL library (on both Unix and Windows) and initialize the plug-in. The only requirement is that the library reside in a directory that appears in the shared library path (`LD_LIBRARY_PATH` on most Unix platforms, `PATH` on Windows).

Instead of dynamically loading the plug-in at runtime your application can also explicitly link with and register the plug-in. To register the plug-in, you must call the `Ice::registerIceSSL(bool loadOnInitialize = true)` function before the communicator initialization. The `loadOnInitialize` parameter specifies if the plug-in is installed when the communicator is initialized. If set to `false`, you will need to enable the plugin by setting the `Ice.Plugin.IceSSL` property to `1`.

Additional configuration properties are usually necessary as well.

> `Ice::registerIceSSL` is a simple helper function that calls `Ice::registerPluginFactory`.

iOS and UWP applications

iOS and UWP applications are using static linking and can't load at runtime the IceSSL plug-in. Instead, they need to link with the IceSSL library and register the plug-in with the `Ice::registerIceSSL(bool loadOnInitialize = true)` function. See Using Plugins with Static Libraries.

OpenSSL on Windows

Ice provides two implementations of IceSSL on Windows, a default implementation that relies on SChannel and a secondary implementation that relies on OpenSSL. To install IceSSL for OpenSSL on Windows, set `Ice.Plugin.IceSSL` as shown below:

```
Ice.Plugin.IceSSL=IceSSLOpenSSL:createIceSSLOpenSSL
```

The plug-in name remains `IceSSL`, and you can only install one IceSSL with a given communicator. You can however create multiple communicators in the same application, each configured with a different IceSSL plugin.

> IceSSL for OpenSSL is not included in the Ice binary distributions for Windows. You need to build Ice C++ from sources to get this IceSSL plug-in.

Java Applications

The format for the `Ice.Plugin` property is shown below:

Java Java Compat

```
Ice.Plugin.IceSSL=IceSSL:com.zeroc.IceSSL.PluginFactory
```

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
```

The IceSSL classes are included in `ice.jar`, therefore no other changes to your `CLASSPATH` are necessary.

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value is the name of a class that allows the Ice run time to initialize the plug-in.

Additional configuration properties are usually necessary as well.

JavaScript applications

SSL is not supported with JavaScript.

.NET Applications

The format for the `Ice.Plugin` property is shown below:

```
Ice.Plugin.IceSSL=IceSSL.dll:IceSSL.PluginFactory
```

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value contains the file name of the IceSSL assembly as well as the name of a class, `IceSSL.PluginFactory`, that allows the Ice run time to initialize the plug-in.

You may also specify a partially or fully qualified assembly name instead of the file name in an `Ice.Plugin` property. For example, you can use the following configuration to load the plug-in from the zeroc.ice.net NuGet package:

.NET Framework 4.5 .NET Core 2.0

```
Ice.Plugin.IceSSL=IceSSL,Version=3.7.0.0,Culture=neutral,PublicKeyToken
=cdd571ade22f2f16:IceSSL.PluginFactory
```

```
Ice.Plugin.IceSSL=IceSSL:IceSSL.PluginFactory
```

With .NET Framework you *must* use a fully qualified assembly name to load the IceSSL plug-in from the Global Assembly Cache.

Additional configuration properties are usually necessary as well.

Objective-C applications

Objective-C applications use the IceSSL C++ plug-in so the configuration is the same as the C++ configuration described above.

If instead of dynamically loading the plug-in at runtime your application prefers to link with the library, it needs to register the plug-in with the `ICEregisterIceSSL(BOOL loadOnInitialize)` function. The `loadOnInitialize` parameter specifies if the plug-in is installed when the communicator is initialized. If set to `NO`, you will need to enable the plugin by setting the `Ice.Plugin.IceSSL` property to `1`.

iOS applications

iOS applications are using static linking and can't load at runtime the IceSSL plug-in. Instead, they need to link with the IceSSL library and register the plug-in with the `ICEregisterIceSSL(BOOL loadOnInitialize)` function. See Using Plugins with Static Libraries.

PHP applications

You should use the same configuration as the C++ configuration described above to install the IceSSL plugin.

Python applications

When using the PyPI package, the IceSSL plugin is automatically installed when the Ice runtime is loaded from the Python module, no additional configuration is required. If you are not using the PyPI package but a build provided with system packages such as RPM or DEB, you should use the same configuration as the C++ configuration described above to install the IceSSL plugin.

Ruby applications

When using the GEM package, the IceSSL plugin is automatically installed when the Ice runtime is loaded from the Ruby module, no additional configuration is required. If you are not using the GEM package but a build provided with system packages such as RPM or DEB, you should use the same configuration as the C++ configuration described above to install the IceSSL plugin.

Creating SSL Endpoints

Installing the IceSSL plug-in enables you to use a new protocol, `ssl`, in your endpoints. For example, the following adapter endpoint list creates a TCP endpoint, an SSL endpoint, and a UDP endpoint:

```
MyAdapter.Endpoints=tcp -p 4063:ssl -p 4064:udp -p 4063
```

As this example demonstrates, it is possible for a UDP endpoint to use the same port number as a TCP or SSL endpoint, because UDP is a different protocol and therefore has an indepdendent set of ports. It is not possible for a TCP endpoint and an SSL endpoint to use the same port number, because SSL is essentially a layer over TCP.

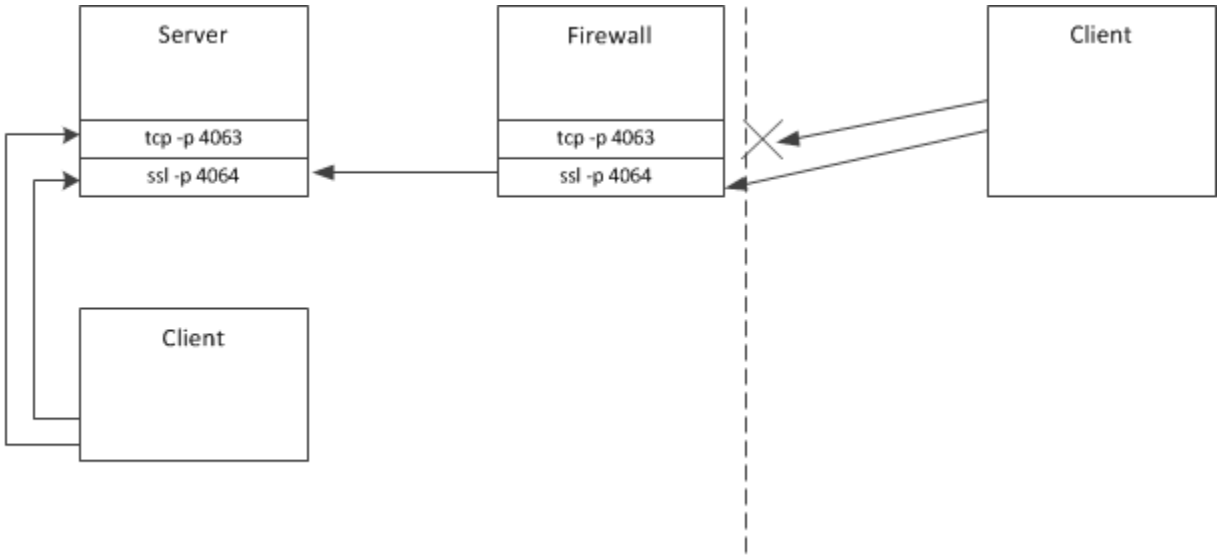
Using SSL in proxy endpoints is equally straightforward:

```
MyProxy=MyObject:tcp -p 4063:ssl -p 4064:udp -p 4063
```

Endpoint Security Considerations

Defining an object adapter's endpoints to use multiple protocols, as shown above, has obvious security implications. If your intent is to use SSL to protect session traffic and/or restrict access to the server, then you should only define SSL endpoints.

There can be situations, however, in which insecure endpoint protocols are advantageous. The figure below illustrates an environment in which TCP endpoints are allowed behind the firewall, but external clients are required to use SSL:



An application of multiple protocol endpoints.

The firewall in the illustration is configured to block external access to TCP port 4063 and to forward connections to port 4064 to the server machine.

One reason for using TCP behind the firewall is that it is more efficient than SSL and requires less administrative work. Of course, this scenario assumes that internal clients can be trusted, which is not true in many environments.

For more information on using SSL in complex network architectures, refer to our discussion of the Glacier2 router.

See Also

- Plug-in Facility
- Configuring IceSSL
- Object Adapter Endpoints
- Proxy Endpoints
- Glacier2
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

Configuring IceSSL

This page describes some general configuration topics:

You can find platform-specific instructions for configuring IceSSL on the following pages:

- Configuring IceSSL for OpenSSL - For Linux applications; also for Windows applications that use OpenSSL instead of the default Schannel
- Configuring IceSSL for Schannel - For Windows and UWP applications in C++, .NET
- Configuring IceSSL for Secure Transport - For macOS and iOS applications
- Configuring IceSSL for Java

Security Considerations for Ciphersuites

A ciphersuite represents a particular combination of encryption, authentication and hashing algorithms. You can configure the ciphersuites that the underlying SSL engines are allowed to negotiate during handshaking with a peer. By default, IceSSL uses the underlying engine's default ciphersuites, but you can define a property to customize the ciphersuite list. Normally the default configuration is chosen to eliminate relatively insecure ciphersuites such as ADH. In general, we recommend setting `IceSSL.Trace.Security=1` when experimenting with ciphersuite configurations. When executing your program, pay special attention to the log output to verify the ciphersuites that IceSSL has enabled, as well as the protocol and ciphersuite negotiated for each connection.

Note that the engine may not negotiate the most secure ciphersuite possible when using its default configuration. We recommend setting `IceSSL.Protocols=TLS1_2` to require the latest TLS version and, if your platform supports it, modifying the allowable ciphersuites to guarantee that an acceptably secure suite will be negotiated. By limiting the possible ciphersuites, you're establishing a security policy that it's preferable for a connection attempt to fail if the remote peer does not support a compatible suite.

Refer to the platform notes in the `IceSSL.Ciphers` property description for the details about your platform.

Configuring Trust Relationships

Declaring that you trust a certificate authority implies that you trust any peer whose certificate was signed directly or indirectly by that certificate authority. It is necessary to use this broad definition of trust in some applications, such as a public Web server. In more controlled environments, it is a good idea to restrict access as much as possible, and IceSSL provides a number of ways for you to do that.

Trusted Peers

After the low-level SSL engine has completed its authentication process, IceSSL can be configured to take additional steps to verify whether a peer should be trusted. The `IceSSL.TrustOnly` family of properties defines a collection of acceptance and rejection filters that IceSSL applies to the distinguished name of a peer's certificate in order to determine whether to allow the connection to proceed. IceSSL permits the connection if the peer's distinguished name matches any of the acceptance filters and does not match any of the rejection filters.

A distinguished name uniquely identifies a person or entity and is generally represented in the following textual form:

```
C=US, ST=Florida, L=Palm Beach Gardens, O="ZeroC, Inc.", OU=Servers,
CN=Quote Server
```

Suppose we are configuring a client to communicate with the server whose distinguished name is shown above. If we know that the client is allowed to communicate only with this server, we can enforce this rule using the following property:

```
IceSSL.TrustOnly=O="ZeroC, Inc.", OU=Servers, CN=Quote Server
```

With this property in place, IceSSL allows a connection to proceed only if the distinguished name in the server's certificate matches this filter.

The property may contain multiple filters, separated by semicolons, if the client needs to communicate with more than one server. Additional variations of the property are also supported.

If the `IceSSL.TrustOnly` properties do not provide the selectivity you require, the next step is to install a custom certificate verifier.

Verification Depth

In order to authenticate a peer, SSL obtains the peer's certificate chain, which includes the peer's certificate as well as that of the root CA. SSL verifies that each certificate in the chain is valid, but there still remains a subtle security risk. Suppose that we have identified a trusted root CA (via its certificate), and a peer has supplied a valid certificate chain signed by our trusted root CA. It is possible for an attacker to obtain a special signing certificate that is signed by our root CA and therefore trusted implicitly. The attacker can use this certificate to sign fraudulent certificates with the goal of masquerading as a trusted peer, presumably for some nefarious purpose.

We could use the `IceSSL.TrustOnly` properties described above in an attempt to defend against such an attack. However, the attacker could easily manufacture a certificate containing a distinguished name that satisfies the trust properties.

If you know that all trusted peers present certificate chains of a certain length, set the property `IceSSL.VerifyDepthMax` so that IceSSL automatically rejects longer chains. The default value of this property is three, therefore you may need to set it to a larger value if you expect peers to present longer chains.

In situations where you cannot make assumptions about the length of a peer's certificate chain, yet you still want to examine the chain before allowing the connection, you should install a custom certificate verifier.

Configuring Secure Proxies

Proxies may contain any combination of secure and insecure endpoints. An application that requires secure communication can guarantee that proxies it manufactures itself, such as those created by calling `stringToProxy`, contain only secure endpoints. However, the application cannot make the same assumption about proxies received as the result of a remote invocation.

The simplest way to guarantee that all proxies use only secure endpoints is to define the `Ice.Override.Secure` configuration property:

```
Ice.Override.Secure=1
```

Setting this property is equivalent to invoking the proxy method `ice_secure(true)` on every proxy. When enabled, attempting to establish a connection using a proxy that does not contain a secure endpoint results in `NoEndpointException`.

If you want the default behavior of proxies to give precedence to secure endpoints, you can set this property instead:

```
Ice.Default.PreferSecure=1
```

Note that proxies may still attempt to establish connections to insecure endpoints, but they try all secure endpoints first. This is equivalent to invoking `ice_preferSecure(true)` on a proxy.

IceSSL Diagnostics

You can use two configuration properties to obtain more information about the plug-in's activities. Setting `IceSSL.Trace.Security=1` enables the plug-in's diagnostic output, which includes a variety of messages regarding events such as ciphersuite selection, peer verification and trust evaluation. The other property, `Ice.Trace.Network`, determines how much information is logged about network events such as connections and packets. Note that the output generated by `Ice.Trace.Network` also includes other transports such as TCP and UDP. Additional platform-specific logging information may also be available. Refer to the appropriate Configuring IceSSL page for your platform for more details.

See Also

- Proxy Methods
- Filtering Proxy Endpoints
- Public Key Infrastructure
- Using IceSSL
- Programming IceSSL

- Advanced IceSSL Topics
- IceSSL.*

Configuring IceSSL for OpenSSL

After installing IceSSL, an application typically needs to define a handful of additional properties to configure settings such as the location of certificate and key files. This page provides an introduction to configuring the plug-in for applications using the OpenSSL version of the plug-in.

On this page:

Configuring IceSSL for OpenSSL on Linux

Our first example shows the properties that are sufficient in many situations:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=cert.pfx
IceSSL.CAs=ca.pem
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your certificate and key files. The two properties that follow it define the files containing the program's certificate with private key and trusted CA certificate, respectively. This example assumes the files contain RSA keys, and IceSSL requires the files to use the Privacy Enhanced Mail (PEM) encoding. Finally, the `IceSSL.Password` property specifies the password of the private key.

> It is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

DSA Example for OpenSSL on Linux

If you used DSA to generate your keys, one additional property is necessary:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=cert_dsa.pfx
IceSSL.CAs=ca.pem
IceSSL.Password=password
IceSSL.Ciphers=DEFAULT:DSS
```

The `IceSSL.Ciphers` property adds support for DSS authentication to the plug-in's default set of ciphersuites.

RSA and DSA Example for OpenSSL on Linux

It is also possible to specify certificates and keys for both RSA and DSA by including two filenames in the `IceSSL.CertFile` property. The file names must be separated using the platform's path separator. The example below demonstrates the configuration:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=cert_rsa.pfx:cert_dsa.pfx
IceSSL.CAs=ca.pem
IceSSL.Password=password
IceSSL.Ciphers=DEFAULT:DSS
```

ADH Example for OpenSSL on Linux

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.Ciphers=ADH:!LOW:!MD5:!EXP:!3DES:@STRENGTH
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH (which is disabled by default) and eliminates low-strength ciphersuites.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

Configuring IceSSL for OpenSSL on Windows

C++ Windows applications can use an alternative implementation of the IceSSL plug-in that uses OpenSSL instead of the default implementation that uses SChannel:

```
Ice.Plugin.IceSSL=IceSSLOpenSSL:createIceSSLOpenSSL

IceSSL.DefaultDir=C:\certs
IceSSL.CAs=cacert.pem
IceSSL.CertFile=cert.pfx
IceSSL.Password=password
```

> Applications using the OpenSSL plug-in cannot use Windows certificate stores.

ADH Example for OpenSSL on Windows

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSLOpenSSL:createIceSSLOpenSSL
IceSSL.Ciphers=ADH:!LOW:!MD5:!EXP:!3DES:@STRENGTH
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH (which is disabled by default) and eliminates low-strength ciphersuites.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

See Also

- Public Key Infrastructure
- Using IceSSL
- Programming IceSSL
- Advanced IceSSL Topics
- IceSSL.*

Configuring IceSSL for Schannel

After installing IceSSL, an application typically needs to define a handful of additional properties to configure settings such as the location of certificate and key files. This page provides an introduction to configuring the plug-in for Windows applications using the Microsoft Secure Channel (Schannel) version of the plug-in.

On this page:

Configuring IceSSL for C++ and .NET on Windows

IceSSL for .NET and C++ supports loading certificates, keys, and trusted CA certificates from regular files. You can also make use of Windows certificate stores.

Using Certificate Files

Our first example demonstrates how to configure IceSSL with certificate files:

```
# For .NET:
Ice.Plugin.IceSSL=IceSSL.dll:IceSSL.PluginFactory
# For C++:
Ice.Plugin.IceSSL=IceSSL:createIceSSL

IceSSL.DefaultDir=C:\certs
IceSSL.CAs=cacert.pem
IceSSL.CertFile=cert.pfx
IceSSL.Password=password
```

Notice that the value for `Ice.Plugin.IceSSL` differs depending on the language mapping you're using. The `Ice.Plugin` properties also support an alternate syntax that allows you to define both variations in the same configuration file if desired.

The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your certificate files. The two subsequent properties identify the files containing the trusted CA certificate and the certificate with private key, respectively. When these properties specify a relative path name, as shown here, IceSSL attempts to find these files in the directory defined by `IceSSL.DefaultDir`. The file specified in `IceSSL.CAs` should normally be in the Privacy Enhanced Mail (PEM) format, whereas the file in `IceSSL.CertFile` must use the Personal Information Exchange (PFX, also known as PKCS#12) format and contain both a certificate and its corresponding private key. The `IceSSL.Password` property specifies the password used to secure the file.

> It is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

Using Certificate Stores

Windows uses certificate stores as the persistent repositories of certificates and keys. Furthermore, there are two distinct sets of certificate stores, one for the current user and another for the local machine.

Managing Certificates with the Microsoft Management Console

On Windows, you can use the Microsoft Management Console (MMC) to browse the contents of the various certificate stores. To start the console, run `MMC.EXE` from a command window, or choose Run from the Start menu and enter `MMC.EXE`.

Once the console is running, you need to install the Certificates "snap-in" by choosing Add/Remove Snap-in from the File menu. Click the

Add button, choose Certificates in the popup window and click Add. If you wish to manage certificates for the current user, select My Current Account and click Finish. To manage certificates for the local computer, select Computer Account and click Next, then select Local Computer and click Finish.

When you have finished adding snap-ins, close the Add Standalone Snap-in window and click OK on the Add/Remove Snap-in window. Your Console Root window now contains a tree structure that you can expand to view the available certificate stores. If you have a certificate in a file that you want to add to a store, click on the desired store, then open the Action menu and select All Tasks/Import.

Configuring IceSSL using Certificate Stores

If the program's certificate and private key are already installed in a certificate store, you can select it using the `IceSSL.FindCert` configuration property as shown in the following example:

```
Ice.Plugin.IceSSL=...
IceSSL.FindCert=subject:"Quote Server"
IceSSL.CertStore=My
IceSSL.CertStoreLocation=LocalMachine
```

An `IceSSL.FindCert` property executes a query in a particular certificate store and selects all of the certificates that match the given criteria. In the example above, the location of the certificate store is `LocalMachine` (as defined by `IceSSL.CertStoreLocation`) and the store's name is `My` (as defined by `IceSSL.CertStore`). When using MMC to browse the certificate stores, this is equivalent to the store "Personal" in the location "Certificates (Local Computer)."

The other legal value for `IceSSL.CertStoreLocation` is `CurrentUser`. The following table shows the valid values for `IceSSL.CertStore` and their equivalents in MMC:

Property Value	MMC Name
`AddressBook`	Other People
`AuthRoot`	Third-Party Root Certification Authorities
`CertificateAuthority`	Intermediate Certification Authorities
`Disallowed`	Untrusted Certificates
`My`	Personal
`Root`	Trusted Root Certification Authorities
`TrustedPeople`	Trusted People
`TrustedPublisher`	Trusted Publishers

The search criteria consists of *name:value* pairs that perform case-insensitive comparisons against the fields of each certificate in the specified store, and the special property value `*` selects every certificate in the store. Typically, however, the criteria should select a single certificate. In a server, IceSSL must supply Windows with the certificate that represents the server's identity; if a configuration matches several certificates, IceSSL chooses one (in an undefined manner) and logs a warning to notify you of the situation.

Selecting a certificate from a store is more secure than using a certificate file via the `IceSSL.CertFile` property because it is not necessary to specify a plain-text password. MMC prompts you for the password when initially importing a certificate into a store, so the password is not required when an application uses that certificate to identify itself.

Trusted CA Certificates

We made no mention of trusted CA certificates in the configuration above. For a program to be able to successfully authenticate a certificate it receives from a peer, all trusted CA certificates must either be loaded via the `IceSSL.CAs` property or already be present in the appropriate certificate stores.

When installing a trusted CA certificate, authentication succeeds only when the certificate is installed into one of the following stores:

- Local Computer / Trusted Root Certification Authorities
- Local Computer / Third-Party Root Certification Authorities
- Current User / Trusted Root Certification Authorities

Note that administrative privileges are required when installing a certificate into a Local Computer store.

Configuring IceSSL for UWP on Windows

IceSSL for UWP supports loading certificates and keys from regular files. You can also use certificates and keys from certificate stores. Trusted CA certificates must be configured using the Trusted Root Certification Authorities.

> The ability to use client-side certificates in UWP is a new feature added in Ice 3.7.

Using Certificate Files

Our first example demonstrates how to configure IceSSL with certificate files:

```
Ice::registerIceSSL();
Ice::InitializationData initData;
initData->properties = Ice::createProperties();

properties->setProperty("IceSSL.CertFile", "ms-appx:///cert.pfx");
properties->setProperty("IceSSL.Password", "password");
```

The file in `IceSSL.CertFile` must use the Personal Information Exchange (PFX, also known as PKCS#12) format and contain both a certificate and its corresponding private key. The file must use `"ms-appx://"` or `"ms-appdata://"` URIs. The `IceSSL.Password` property specifies the password used to secure the file.

> It is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

Using Certificate Stores

Configuring IceSSL using Certificate Stores

If the program's certificate and private key are already installed in a certificate store, you can select it using the `IceSSL.FindCert` configuration property as shown in the following example:

```
IceSSL.FindCert=FRIENDLYNAME:"Client"
IceSSL.CertStore=My
```

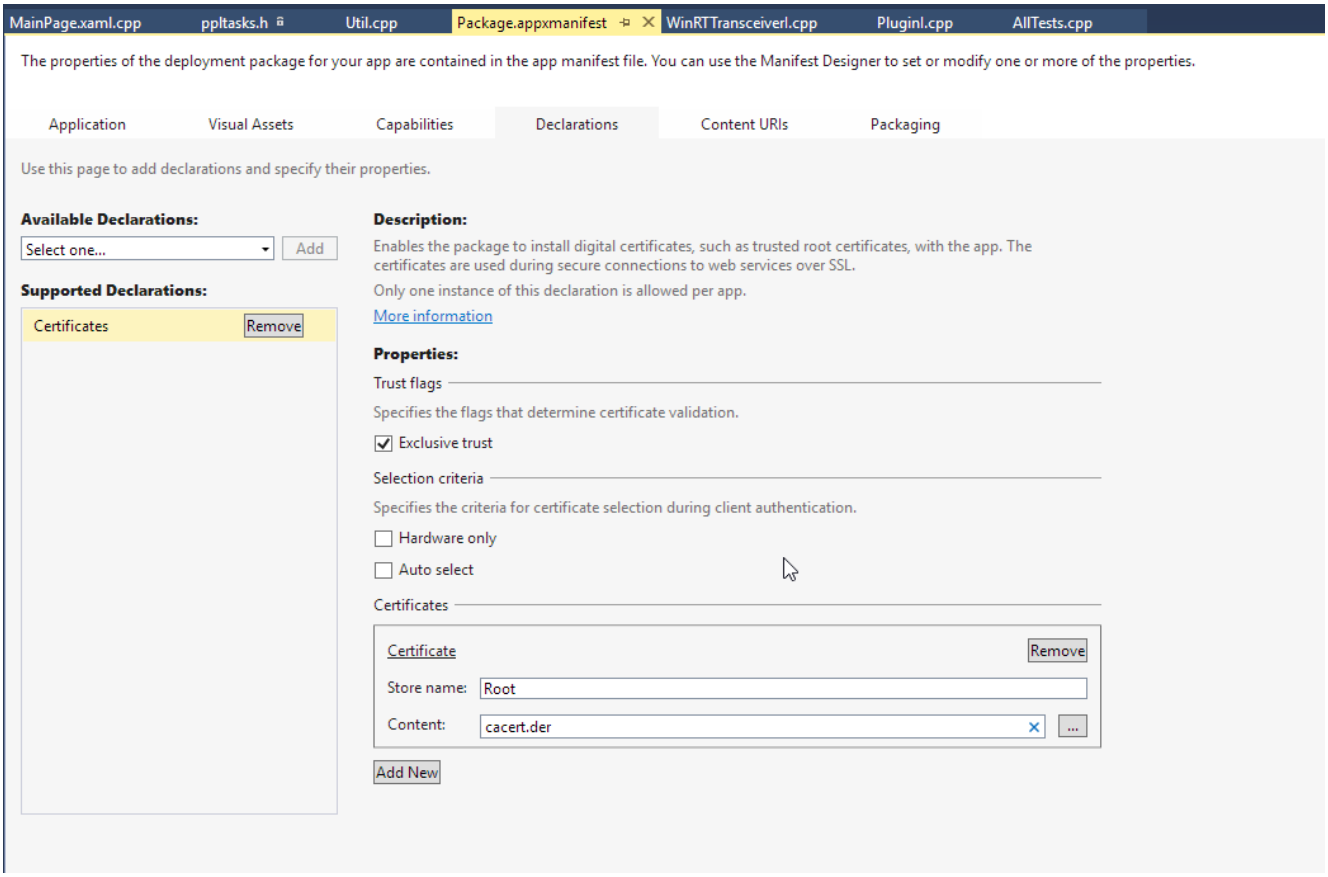
An `IceSSL.FindCert` property executes a query in all certificate stores with the given name and selects all of the certificates that match the given criteria. In the example above, the store's name is `My` (as defined by `IceSSL.CertStore`). this is equivalent to the store "Personal". If the UWP application manifest has set the "Shared User Certificates" capability the query will look into the current user and application "Personal" stores otherwise it will only match certificates in the application "Personal" store.

> UWP applications can use the application certificate store or the system certificate stores. Application stores are isolated from other system stores.

Trusted CA Certificates

UWP application by default trust certificate authorities from the system and application "Trusted Root Certification Authorities" stores. To limit the trusted authorities to the application "Trusted Root Certification Authorities" store you must use the "Certificates" declaration in the application manifest and set the "Exclusive Trust" flag. The application "Trusted Root Certification Authorities" store is empty by default

empty. The "Certificates" declaration in the application manifest allows you to add new certificates. The certificates must be DER encoded.



IceSSL Diagnostics for .NET

You can use two configuration properties to obtain more information about the plug-in's activities. Setting `IceSSL.Trace.Security=1` enables the plug-in's diagnostic output, which includes a variety of messages regarding events such as ciphersuite selection, peer verification and trust evaluation. The other property, `Ice.Trace.Network`, determines how much information is logged about network events such as connections and packets. Note that the output generated by `Ice.Trace.Network` also includes other transports such as TCP and UDP.

You can enable additional tracing output by creating an XML file such as the one shown below:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <system.diagnostics>
        <trace autoflush="true"/>
        <sources>
            <source name="System.Net">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
            <source name="System.Net.Sockets">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
            <source name="System.Net.Cache">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
        </sources>
        <sharedListeners>
            <add
            name="System.Net"
            type="System.Diagnostics.TextWriterTraceListener"
            initializeData="trace.txt"
            />
        </sharedListeners>
        <switches>
            <add name="System.Net" value="Verbose"/>
            <add name="System.Net.Sockets" value="Verbose"/>

            <add name="System.Net.Cache" value="Verbose"/>
        </switches>
    </system.diagnostics>
</configuration>
```

In this example, the output is stored in the file `trace.txt`. To activate tracing, give the XML file the same name as your executable with a `.config` extension (such as `server.exe.config`), and place it in the same directory as the executable.

See Also

- Public Key Infrastructure
- Using IceSSL
- Programming IceSSL
- Advanced IceSSL Topics
- IceSSL.*

Configuring IceSSL for Secure Transport

After installing IceSSL, an application typically needs to define a handful of additional properties to configure settings such as the location of certificate and key files. This page provides an introduction to configuring the plug-in for iOS and macOS applications.

On this page:

Configuring IceSSL for macOS

Our first example shows the properties that are sufficient in many situations:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=cert.pfx
IceSSL.CAs=ca.pem
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your certificate files. The properties that follow it define the files containing the program's certificate with private key, and trusted CA certificate, respectively. Finally, the `IceSSL.Password` property specifies the password necessary to open `cert.pfx`.

> It is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

Keychain Examples for macOS

IceSSL imports the certificate specified by `IceSSL.CertFile` into a keychain. IceSSL uses the user's default keychain unless you choose a different one by defining the `IceSSL.Keychain` property:

```
IceSSL.Keychain=Test Keychain
```

If the specified keychain file does not exist, IceSSL will create it. The file name is opened relative to the user's current working directory unless an absolute path name is provided.

The user's default keychain is normally unlocked after logging into the system, so IceSSL doesn't usually require a password to import a certificate into this keychain. However, if your keychain is not unlocked automatically, or if you've selected a different keychain, you can supply a password using the `IceSSL.KeychainPassword` property:

```
IceSSL.KeychainPassword=password
```

> It is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password.

If you don't define `IceSSL.KeychainPassword` and a password is required to open the keychain, macOS will prompt the user for the password.

To use a certificate that's already in a keychain, you can omit the `IceSSL.CertFile` and `IceSSL.Password` properties and use `IceSSL.FindCert` instead:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CAs=ca.pem
IceSSL.FindCert=Label:Client
```

Here we've instructed IceSSL to locate the certificate in the default keychain labeled `Client` and use that as the program's identity.

ADH Example for macOS

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.Ciphers=(DH_anon*)
IceSSL.DHParams=dhparams.der
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH. We also recommend setting `IceSSL.DHParams` with the name of a DER-encoded file containing pre-generated DH parameters.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

Configuring IceSSL for iOS

With iOS applications, certificate files are loaded from the application's resource bundle. If the application's target platform is macOS, certificate files can also be loaded directly from the file system. Consider the following properties:

```
IceSSL.DefaultDir=certs
IceSSL.CAs=cacert.der
IceSSL.CertFile=cert.pfx
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the location of your certificate files. Defining `IceSSL.DefaultDir` means IceSSL searches for certificate files relative to the specified directory. For the properties in the example above, IceSSL composes the pathnames `certs/cacert.der` and `certs/cert.pfx`. If `IceSSL.DefaultDir` is not defined, IceSSL uses the certificate file pathnames exactly as they are supplied.

As mentioned earlier, IceSSL has different semantics for locating certificate files depending on the target platform. For the iPhone and iPhone simulator, IceSSL attempts to open a certificate file in the application's resource bundle as `Resources/`*`DefaultDir/file`* if `IceSSL.DefaultDir` is defined, or as simply `Resources/`*`file`* otherwise. If the target platform is macOS and the certificate file cannot be found in the resource bundle, IceSSL also attempts to open the file in the file system as *`DefaultDir/file`* if a default directory is specified, or as simply *`file`* otherwise.

IceSSL requires that the CA certificate file specified by `IceSSL.CAs` use the DER format. The certificate file in `IceSSL.CertFile` must use the Personal Information Exchange (PFX, also known as PKCS#12) format and contain both a certificate and its corresponding private key. The `IceSSL.Password` property specifies the password used to secure the certificate file.

Keychain Examples for iOS

IceSSL imports the certificate specified by `IceSSL.CertFile` into a keychain. IceSSL uses the `login` keychain by default unless you choose a different one by defining the `IceSSL.Keychain` property:

```
IceSSL.Keychain=Test Keychain
```

The `login` keychain is the user's default keychain, which is normally unlocked after logging into the system. IceSSL does not usually require a password to import a certificate into the `login` keychain. However, if your `login` keychain is not unlocked automatically, or if you have selected a different keychain, you can supply a password using the `IceSSL.KeychainPassword` property:

```
IceSSL.KeychainPassword=password
```

IceSSL Diagnostics for macOS

You can use two configuration properties to obtain more information about the plug-in's activities. Setting `IceSSL.Trace.Security=1` enables the plug-in's diagnostic output, which includes a variety of messages regarding events such as ciphersuite selection, peer verification and trust evaluation. The other property, `Ice.Trace.Network`, determines how much information is logged about network events such as connections and packets. Note that the output generated by `Ice.Trace.Network` also includes other transports such as TCP and UDP.

macOS also includes a TLS logging facility. To enable it, run the following command in Terminal:

```
$ sudo defaults write /Library/Preferences/com.apple.security
SSLDebugScope -bool true
```

To see the log entries you must enable logging for warnings in syslog:

```
$ sudo syslog -c 0 -w
```

Then run syslog in 'wait' mode:

```
$ syslog -w
```

See Also

- Public Key Infrastructure
- Using IceSSL
- Programming IceSSL
- Advanced IceSSL Topics
- IceSSL.*

Configuring IceSSL for Java

After installing IceSSL, an application typically needs to define a handful of additional properties to configure settings such as the location of certificate and key files. This page provides an introduction to configuring the plug-in for Java applications.

On this page:

Configuring Keystores for Java

IceSSL uses Java's native format for storing keys and certificates: the keystore.

A keystore is represented as a file containing key pairs and associated certificates, and is usually administered using the `keytool` utility supplied with the Java run time. Keystores serve two roles in Java's SSL architecture:

1. A keystore containing a key pair identifies the peer and is usually closely guarded.
2. A keystore containing public certificates represents the identities of trusted peers and can be freely shared. These keystores are also referred to as "truststores" when they are used to store only trusted certificate chains.

A single keystore file can fulfill both of these purposes.

Java supports a pluggable architecture for keystore implementations in which a system property selects a particular implementation as the default keystore type. IceSSL uses the default keystore type unless otherwise specified.

A password is assigned to each key pair in a keystore, as well as to the keystore itself. IceSSL must be provided with the password for the key pair, but the keystore password is optional. If a keystore password is specified, it is used only to verify the keystore's integrity. IceSSL requires that all of the key pairs in a keystore have the same password.

Our first example shows the properties that are sufficient in many situations:

```
Ice.Plugin.IceSSL=IceSSL:com.zeroc.IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Keystore=keys.jks
IceSSL.Truststore=ca.jks
IceSSL.Password=password
```

IceSSL resolves the filenames defined in its configuration properties as follows:

1. Attempt to open the file as a class loader resource. This is especially useful for deploying applications with special security restrictions, such as applets.
2. Attempt to open the file in the local file system.
3. If `IceSSL.DefaultDir` is defined, prepend its value and try steps 1 and 2 again. The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your keystore and truststore files.

The `IceSSL.Password` property specifies the password of the key pair.

> It is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. IceSSL also supports alternate ways to supply a password.

DSA Example for Java

Java supports both RSA and DSA keys. No additional properties are necessary to use DSA:

```
Ice.Plugin.IceSSL=IceSSL:com.zeroc.IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Keystore=dsakeys.jks
IceSSL.Truststore=ca.jks
IceSSL.Password=password
```

ADH Example for Java

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSL:com.zeroc.IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Ciphers=NONE (DH_anon.*AES)
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH, which is disabled by default. Furthermore, this setting enables only those ADH ciphersuites that support AES encryption and eliminates other, lower-strength ciphersuites that may not be supported by recent JVMs.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

Configuring Ciphersuites for Java

A ciphersuite represents a particular combination of encryption, authentication and hashing algorithms. You can configure the ciphersuites that the underlying SSL engines are allowed to negotiate during handshaking with a peer. By default, IceSSL uses the underlying engine's default ciphersuites, but you can define a property to customize the ciphersuite list. Normally the default configuration is chosen to eliminate relatively insecure ciphersuites such as ADH, which is why it needs to be explicitly enabled as we saw in the example above.

IceSSL for Java interprets the value of `IceSSL.Ciphers` as a sequence of expressions that filter the selected ciphersuites using name and pattern matching. If the property is not defined, the Java security provider's default ciphersuites are used. The following table defines the valid expressions that may appear in the property value.

Expression	Description
`NONE`	Disables all ciphersuites. If specified, it must appear first.
`ALL`	Enables all supported ciphersuites. If specified, it must appear first. This expression should be used with caution, as it may enable low-security ciphersuites.
`NAME`	Enables the ciphersuite matching the given name.
`!`*`NAME`*	Disables the ciphersuite matching the given name.
`(EXP)`	Enables ciphersuites whose names contain the regular expression *EXP*.
`!(EXP)`	Disables ciphersuites whose names contain the regular expression *EXP*.

To determine the set of enabled ciphersuites, the plug-in begins with a list of ciphersuite names containing the default set as determined by the security provider. The expressions in the property value add and remove ciphersuites from this list and are evaluated in the order of appearance. For example, consider the following property definition:

```
IceSSL.Ciphers=NONE (RSA.*AES) !(EXPORT)
```

The expressions in this property have the following effects:

- `NONE` clears the list of enabled ciphersuites.
- `(RSA.*AES)` is a regular expression that enables ciphersuites whose names contain the string "RSA" followed by "AES", meaning ciphersuites using RSA authentication and AES encryption.
- `!(EXPORT)` is a regular expression that disables any of the selected ciphersuites whose names contain the string "EXPORT", meaning ciphersuites having export-quality strength.

As another example, this property adds anonymous Diffie-Hellman to the default set of ciphersuites and disables export ciphersuites:

```
IceSSL.Client.Ciphers=(DH_anon) !(EXPORT)
```

Finally, this example selects only one ciphersuite:

```
IceSSL.Client.Ciphers=NONE SSL_RSA_WITH_RC4_128_SHA
```

> We recommend setting `IceSSL.Trace.Security=1` when experimenting with ciphersuite configurations. Pay special attention to the log output to verify the ciphersuites that IceSSL has enabled.

IceSSL Diagnostics for Java

You can use two configuration properties to obtain more information about the plug-in's activities. Setting `IceSSL.Trace.Security=1` enables the plug-in's diagnostic output, which includes a variety of messages regarding events such as ciphersuite selection, peer verification and trust evaluation. The other property, `Ice.Trace.Network`, determines how much information is logged about network events such as connections and packets. Note that the output generated by `Ice.Trace.Network` also includes other transports such as TCP and UDP.

You can also use a system property that displays a great deal of information about SSL certificates and connections, including the ciphersuite that is selected for use by each connection. For example, the following command sets the system property that activates the diagnostics:

```
$ java -Djavax.net.debug=ssl MyProgram
```

See Also

- Public Key Infrastructure
- Using IceSSL
- Programming IceSSL
- Advanced IceSSL Topics
- IceSSL.*

Programming IceSSL

The IceSSL configuration properties are flexible enough to satisfy the requirements of many applications, and IceSSL supports a public API that offers even more functionality for those applications that need it.

Topics

- Programming IceSSL in C++
- Programming IceSSL in Java
- Programming IceSSL in .NET
- Programming IceSSL in Other Languages

See Also

- Configuring IceSSL

Programming IceSSL in C++

This page describes the C++ API for the IceSSL plug-in.

On this page:

The IceSSL Plugin Interface in C++

The IceSSL C++ plug-in has several implementations listed in the table below. Its API consists of implementation-independent classes directly in the `IceSSL` namespace plus implementation-dependent classes in nested namespaces such as `IceSSL::OpenSSL`.

Underlying SSL/TLS Implementation	IceSSL Plug-in Class	IceSSL Certificate Class	Platform Availability
OpenSSL	`IceSSL::Plugin` `IceSSL::OpenSSL::Plugin`	`IceSSL::Certificate` `IceSSL::OpenSSL::Certificate`	Linux (default), Windows
SecureTransport	`IceSSL::Plugin`	`IceSSL::Certificate` `IceSSL::SecureTransport::Certificate`	macOS (default)
SChannel	`IceSSL::Plugin`	`IceSSL::Certificate` `IceSSL::SChannel::Certificate`	Windows (default)
UWP	`IceSSL::Plugin`	`IceSSL::Certificate` `IceSSL::UWP::Certificate`	UWP (default)

Each platform has a default implementation, indicated with (default) in the table above. Static functions on the `IceSSL::Certificate` class are supplied by the default platform-dependent implementation.

Applications can interact directly with the IceSSL plug-in using the C++ class `IceSSL::Plugin`. You can retrieve this `Plugin` object as shown below:

C++11 C++98

```
auto communicator = ...
auto pluginMgr = communicator->getPluginManager();
auto plugin = plugin->getPlugin("IceSSL");
auto sslPlugin = std::dynamic_pointer_cast<IceSSL::Plugin>(plugin);
```

```
Ice::CommunicatorPtr communicator = ...
Ice::PluginManagerPtr pluginMgr = communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin = IceSSL::PluginPtr::dynamicCast(plugin);
```

The `Plugin` class provides the following functions:

C++11 C++98

```
namespace IceSSL
{
    class Plugin : public Ice::Plugin
    {
    public:
        virtual void setCertificateVerifier(std::function<bool(const
std::shared_ptr<IceSSL::ConnectionInfo>&)>) = 0;
        virtual void setPasswordPrompt(std::function<std::string()>) =
0;
        virtual std::shared_ptr<Certificate> load(const std::string&)
const = 0;
        virtual std::shared_ptr<Certificate> decode(const std::string&)
const = 0;
    };
}
```

```
// Only with OpenSSL implementation
//
namespace Ice
{
    namespace OpenSSL
    {
        class Plugin : public IceSSL::Plugin
        {
        public:
            virtual Ice::Long getOpenSSLVersion() const = 0;
            virtual void setContext(SSL_CTX*) = 0;
            virtual SSL_CTX* getContext() = 0;
        };
    }
}
```

```
namespace IceSSL
{
    class Plugin : public Ice::Plugin
    {
    public:
        virtual void setCertificateVerifier(const
CertificateVerifierPtr&) = 0;
        virtual void setPasswordPrompt(const PasswordPromptPtr&) = 0;
        virtual CertificatePtr load(const std::string&) const = 0;
        virtual CertificatePtr decode(const std::string&) const = 0;
    };
}
```

```
// Only with OpenSSL implementation
//
namespace Ice
{
    namespace OpenSSL
    {
        class Plugin : public IceSSL::Plugin
        {
        public:
            virtual Ice::Long getOpenSSLVersion() const = 0;
            virtual void setContext(SSL_CTX*) = 0;
            virtual SSL_CTX* getContext() = 0;
        };
    }
}
```

The `setCertificateVerifier` function installs a custom certificate verifier object that the plug-in invokes for each new connection. The `setPasswordPrompt` function provides an alternate way to supply IceSSL with passwords. We discuss certificate verifiers below and revisit the other functions in our discussion of advanced IceSSL programming.

The `load` function creates a `Certificate` object from a file in PEM encoded format. Likewise, the `decode` function creates a certificate from a string in the PEM encoding format. The most-derived type of the certificate object depends on the plug-in implementation you're using:

Plug-in	Certificate
OpenSSL	`IceSSL::OpenSSL::Certificate`
SChannel	`IceSSL::SChannel::Certificiate`
SecureTransport	`IceSSL::SecureTransport::Certificate`
UWP	`IceSSL::UWP::Certificate`

`IceSSL::OpenSSL::Plugin` extends the base `IceSSL::Plugin` class and provides the `getOpenSSLVersion`, `setContext` and `getContext` functions that are specific to the OpenSSL implementation and are rarely used in practice. The `IceSSL.InitOpenSSL` property is typically used in conjunction with `setContext`.

Obtaining SSL Connection Information in C++

You can obtain information about any SSL connection using the `getInfo` operation on a `Connection` object. IceSSL defines the following type in Slice:

Slice

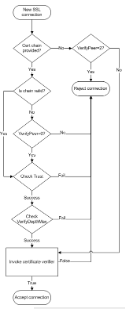
```
// from Ice/Connection.ice
module Ice
{
    local class ConnectionInfo
    {
        ConnectionInfo underlying;
        bool incoming;
        string adapterName;
        string connectionId;
    }
}

// from IceSSL/ConnectionInfo.ice
module IceSSL
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string cipher;
        ["cpp:type:std::vector<CertificatePtr>",
         "java:type:java.security.cert.Certificate[]",

"cs:type:System.Security.Cryptography.X509Certificates.X509Certificate2
[]"]
        Ice::StringSeq certs;
        bool verified;
    }
}
```

The `certs` member contains the peer's certificate chain as a sequence of `IceSSL::CertificatePtr` objects. The `cpp:type` metadata changes the default mapping of the `certs` member to that native type. The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The `verified` member indicates whether IceSSL was able to successfully verify the peer's certificate.

The inherited `underlying` data member contains the connection information of the underlying transport (if SSL is based on TCP, this member will contain an instance of `Ice::TCPEndpointInfo` which you can use to retrieve the remote and local addresses). The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). The `connectionId` data member matches the connection identifier set on the proxy. Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

Installing a Certificate Verifier in C++

A new connection undergoes a series of verification steps before an application is allowed to use it. The low-level SSL engine executes certificate validation procedures and, assuming the certificate chain is successfully validated, IceSSL performs additional verification as directed by its configuration properties. If a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to accept or reject the connection. The value of the `IceSSL.VerifyPeer` property also plays an important role here. We've summarized the process in the following flow chart:



C++11 C++98

With the C++11 mapping, the certificate verifier is a `std::function` provided to the IceSSL plug-in's `setCertificateVerifier` member function:

```
 virtual void setCertificateVerifier(std::function<bool(const
std::shared_ptr<IceSSL::ConnectionInfo>&)>) = 0;
```

IceSSL rejects the connection if the certificate verifier function returns `false`, and allows it to proceed if the function returns `true`. The `verify` function receives a `IceSSL::ConnectionInfo` object that describes the connection's attributes.

The following code demonstrates a simple implementation of a certificate verifier:

```
auto verifier = [](const shared_ptr<IceSSL::ConnectionInfo>& info)
    {
        if(!info->certs.empty())
        {
            auto cert = IceSSL::Certificate::decode(info->certs[0]);
            auto dn = cert->getIssuerDN();
            transform(dn.begin(), dn.end(), dn.begin(), ::tolower);
            if(dn.find("zeroc") != string::npos)
            {
                return true;
            }
        }
        return false;
    };
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

```
shared_ptr<IceSSL::Plugin> sslPlugin = ...
sslPlugin->setCertificateVerifier(verifier);
```

With the C++98 mapping, you must provide an object that implements the `CertificateVerifier` abstract base class:

```
namespace IceSSL
{
    class CertificateVerifier : public IceUtil::Shared
    {
    public:

        virtual bool verify(const IceSSL::ConnectionInfoPtr&) = 0;
    };
    typedef IceUtil::Handle<CertificateVerifier> CertificateVerifierPtr;
}
```

IceSSL rejects the connection if the certificate verifier function returns `false`, and allows it to proceed if the function returns `true`. The `verify` function receives a `IceSSL::ConnectionInfo` object that describes the connection's attributes.

The following class is a simple implementation of a certificate verifier:

```
class Verifier : public IceSSL::CertificateVerifier
{
public:

    bool verify(const IceSSL::ConnectionInfoPtr& info)
    {
        if(!info->certs.empty())
        {
            IceSSL::CertificatePtr cert =
IceSSL::Certificate::decode(info->certs[0]);
            string dn = cert->getIssuerDN();
            transform(dn.begin(), dn.end(), dn.begin(), ::tolower);
            if(dn.find("zeroc") != string::npos)
            {
                return true;
            }
        }
        return false;
    }
};
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

```
IceSSL::PluginPtr sslPlugin = ...
sslPlugin->setCertificateVerifier(verifier);
```

You should install the verifier before any SSL connections are established.

You can also install a certificate verifier using a custom plug-in to avoid making changes to the code of an existing application.

The Ice run time calls the certificate verifier during the connection-establishment process, therefore delays in the certificate verifier implementation have a direct impact on the performance of the application. Do not make remote invocations from your implementation of the certificate verifier.

Using Certificates in C++

The `IceSSL::ConnectionInfo` class contains a vector of `IceSSL::Certificate` objects representing the peer's certificate chain. `IceSSL::Certificate` is a reference-counted convenience class that hides the complexity of the platform's native SSL API inspired by the Java class `X509Certificate`. The `IceSSL::Certificate` class defines the common API for all IceSSL implementations, and APIs that use platform-dependent types are provided by extensions of this class. When a common method is not supported by a SSL implementation it raises `Ice::FeatureNotSupportedException`.

C++11 C++98

```
namespace IceSSL
{
    class Certificate : public std::enable_shared_from_this<Certificate>
    {
    public:

        virtual bool verify(const std::shared_ptr<Certificate>&) const =
0;
        virtual std::string encode() const = 0;

        virtual bool checkValidity() const = 0;
        virtual bool checkValidity(const
std::chrono::system_clock::time_point&) const = 0;
        virtual std::chrono::system_clock::time_point getNotAfter()
const = 0;
        virtual std::chrono::system_clock::time_point getNotBefore()
const = 0;

        virtual std::string getSerialNumber() const = 0;

        virtual DistinguishedName getIssuerDN() const = 0;
        virtual std::vector<std::pair<int, std::string> >
getIssuerAlternativeNames() const = 0;

        virtual DistinguishedName getSubjectDN() const = 0;
        virtual std::vector<std::pair<int, std::string> >
getSubjectAlternativeNames() const = 0;

        virtual int getVersion() const = 0;
        virtual std::string toString() const = 0;

        virtual std::vector<std::shared_ptr<X509Extension>>
getX509Extensions() const = 0;
        virtual std::shared_ptr<X509Extension> getX509Extension(const
std::string&) const = 0;

        virtual std::vector<Ice::Byte> getAuthorityKeyIdentifier() const
= 0;
        virtual std::vector<Ice::Byte> getSubjectKeyIdentifier() const =
0;

        static std::shared_ptr<Certificate> load(const std::string&);
        static std::shared_ptr<Certificate> decode(const std::string&);
    };
}
```

```
namespace IceSSL
{
    class Certificate : public virtual IceUtil::Shared
    {
    public:

        virtual bool verify(const CertificatePtr&) const = 0;
        virtual std::string encode() const = 0;
        virtual bool checkValidity() const = 0;

        virtual bool checkValidity(const IceUtil::Time&) const = 0;

        virtual IceUtil::Time getNotAfter() const = 0;
        virtual IceUtil::Time getNotBefore() const = 0;

        virtual std::string getSerialNumber() const = 0;

        virtual DistinguishedName getIssuerDN() const = 0;
        virtual std::vector<std::pair<int, std::string> >
getIssuerAlternativeNames() const = 0;

        virtual DistinguishedName getSubjectDN() const = 0;
        virtual std::vector<std::pair<int, std::string> >
getSubjectAlternativeNames() const = 0;

        virtual int getVersion() const = 0;
        virtual std::string toString() const = 0;

        virtual std::vector<X509ExtensionPtr> getX509Extensions() const
= 0;
        virtual X509ExtensionPtr getX509Extension(const std::string&)
const = 0;

        virtual std::vector<Ice::Byte> getAuthorityKeyIdentifier() const
= 0;
        virtual std::vector<Ice::Byte> getSubjectKeyIdentifier() const =
0;

        static CertificatePtr load(const std::string&);
        static CertificatePtr decode(const std::string&);
    };
}
```

The more commonly-used functions are described below; refer to the documentation in `IceSSL/Plugin.h` for information on the functions that are not covered.

The static function `load` creates a certificate from the contents of a certificate file PEM-encoded. If an error occurs, the function raises `IceSSL::CertificateReadException`; the reason member provides a description of the problem.

UWP
The UWP implementation of load uses GetFileFromApplicationUriAsync and the file path must use an appropriate URI format.

Use `decode` to obtain a certificate from a PEM-encoded string representing a certificate. The caller must be prepared to catch `IceSSL::CertificateEncodingException` if `decode` fails; the reason member provides a description of the problem.

Both `load` and `decode` return a certificate using the default `IceSSL::Certificate` implementation:

- `IceSSL::SChannel::Certificate` on Windows
- `IceSSL::SecureTransport::Certificate` on macOS
- `IceSSL::UWP::Certificate` for UWP
- `IceSSL::OpenSSL::Certificate` for OpenSSL

All of these classes derive from `IceSSL::Certificate` to provide platform-dependent methods.

The `encode` function creates a PEM-encoded string that represents the certificate. The return value can later be passed to `decode` to recreate the certificate.

The `checkValidity` functions determine whether the certificate is valid. The overloading with no arguments returns true if the certificate is valid at the current time; the other overloading accepts a `system_clock::time_point` with the C++11 mapping (`IceUtil::Time` with the C++98 mapping) and returns true if the certificate is valid at the given time.

The `getNotAfter` and `getNotBefore` functions return the times that define the certificate's valid period, as a `system_clock::time_point` with the C++11 mapping and as a `IceUtil::Time` with the C++98 mapping.

The functions `getIssuerDN` and `getSubjectDN` supply the distinguished names of the certificate's issuer (i.e., the CA that signed the certificate) and subject (i.e., the person or entity to which the certificate was issued). The functions return instances of the class `IceSSL::DistinguishedName`, another convenience class that is described below.

The `getX509Extensions` and `getX509Extension` functions can be used to retrieve a list of the X509 extensions included in the certificate or a specific X509 extension by providing its `OID`, respectively. These methods are only supported by the OpenSSL and SChannel implementations. We discuss extensions further below.

The `getAuthorityKeyIdentifier` and `getSubjectKeyIdentifier` functions return the authority key identifier and subject key identifier respectively as a sequence of bytes. These methods are not supported with UWP or with SecureTransport on iOS.

Finally, the `toString` function returns a human-readable string describing the certificate.

Interacting with the Native Certificates

It is possible to interact with the SSL implementation's native certificates. Tthese APIs are platform-dependent and are defined in extensions of the `IceSSL::Certificate` class. Refer to the header file for each implementation to get more details. The following table summarizes the header files specific to each implementation:

SSL Implementation	Header File
OpenSSL	`IceSSL/OpenSSL.h`
SecureTransport	`IceSSL/SecureTransport.h`
SChannel	`IceSSL/SChannel.h`
UWP	`IceSSL/UWP.h`

Usually you don't need to include these header files in your application; you'll just include `IceSSL/IceSSL.h` which includes the appropriate header files for your platform's default implementation. If you are using the OpenSSL implementation on Windows and need to access OpenSSL-specific APIs, you must include `IceSSL/OpenSSL.h`.

Using Distinguished Names in C++

X.509 certificates use a distinguished name to identify a person or entity. The name is an ordered sequence of relative distinguished names that supply values for fields such as common name, organization, state, and country. Distinguished names are commonly displayed in stringified form according to the rules specified by RFC 2253, as shown in the following example:

```
C=US, ST=Florida, L=Palm Beach Gardens, O="ZeroC, Inc.", OU=Servers,
CN=Quote Server
```

`DistinguishedName` is a convenience class provided by IceSSL to simplify the tasks of parsing, formatting and comparing distinguished names.

C++

```
namespace IceSSL
{
    class DistinguishedName
    {
    public:

        explicit DistinguishedName(const std::string&);
        explicit DistinguishedName(const
std::list<std::pair<std::string, std::string>>&);

        bool match(const DistinguishedName&) const;
        bool match(const std::string&) const;

        operator std::string() const;

        friend bool operator==(const DistinguishedName&, const
DistinguishedName&);
        friend bool operator<(const DistinguishedName&, const
DistinguishedName&);
    };

    inline bool operator!=(const DistinguishedName&, const
DistinguishedName&) { ... }
    // etc., provides all comparison operators.
}
```

The first overloaded constructor accepts a string argument representing a distinguished name encoded using the rules set forth in RFC 2253. The new `DistinguishedName` instance preserves the order of the relative distinguished names in the string. The caller must be prepared to catch `IceSSL::ParseException` if an error occurs during parsing.

The second overloaded constructor requires a list of type-value pairs representing the relative distinguished names. The new `DistinguishedName` instance preserves the order of the relative distinguished names in the list.

The `match` functions perform a partial comparison that does not consider the order of relative distinguished names. Given two instances N1 and N2 of `DistinguishedName`, `N1.match(N2)` returns true if all of the relative distinguished names in `N2` are present in `N1`.

Finally, the string conversion operator encodes the distinguished name in the format described by RFC 2253.

The comparison operators perform an exact match of distinguished names in which the order of the relative distinguished names is important. For two distinguished names to be equal, they must have the same relative distinguished names in the same order.

Using Certificate Extensions in C++

The `Certificate` class provides the methods `getX509Extensions` and `getX509Extension` for obtaining information about its X509

extensions.

> These methods are only supported when using the OpenSSL or Schannel implementations.

The methods return instances of `IceSSL::X509Extension`:

C++11 C++98

```
namespace IceSSL
{
    class X509Extension
    {
    public:

        virtual bool isCritical() const = 0;
        virtual std::string getOID() const = 0;
        virtual std::vector<Ice::Byte> getData() const = 0;
    };
}
```

```
namespace IceSSL
{
    class X509Extension : public virtual IceUtil::Shared
    {
    public:

        virtual bool isCritical() const = 0;
        virtual std::string getOID() const = 0;
        virtual std::vector<Ice::Byte> getData() const = 0;
    };
}
```

An OID is represented as a string. For example, the OID for the subject alternative name extension is `"2.5.29.17"`. The data associated with the extension is provided as a vector of bytes.

The following code demonstrates how to retrieve an extension:

C++11 C++98

```
shared_ptr<IceSSL::Certificate> cert = ...;
shared_ptr<IceSSL::X509Extension> subjectAltName =
cert->getX509Extension("2.5.29.17");
if(subjectAltName)
{
    vector<Ice::Byte> data = subjectAltName->getData();
    ...
}
```

```
IceSSL::CertificatePtr cert = ...;
IceSSL::X509ExtensionPtr subjectAltName =
cert->getX509Extension("2.5.29.17");
if(subjectAltName)
{
    vector<Ice::Byte> data = subjectAltName->getData();
    ...
}
```

See Also

- Using Connections
- Public Key Infrastructure
- Configuring IceSSL
- Advanced IceSSL Topics

Programming IceSSL in Java

This page describes the Java API for the IceSSL plug-in.

On this page:

The IceSSL Plugin Interface in Java

Applications can interact directly with the IceSSL plug-in using a native Java interface. A reference to a `Plugin` object must be obtained from the communicator in which the plug-in is installed:

Java Java Compat

```
import com.zeroc.Ice.*;
...
Communicator comm = // ...
PluginManager pluginMgr = comm.getPluginManager();
Plugin plugin = pluginMgr.getPlugin("IceSSL");
com.zeroc.IceSSL.Plugin sslPlugin = (com.zeroc.IceSSL.Plugin)plugin;
```

```
import Ice.*;
...
Communicator comm = // ...
PluginManager pluginMgr = comm.getPluginManager();
Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
```

The `Plugin` interface supports the following methods:

Java Java Compat

```
package com.zeroc.IceSSL;

public interface Plugin extends com.zeroc.Ice.Plugin
{
    void setContext(javax.net.ssl.SSLContext context);
    javax.net.ssl.SSLContext getContext();

    void setCertificateVerifier(CertificateVerifier verifier);
    CertificateVerifier getCertificateVerifier();

    void setPasswordCallback(PasswordCallback callback);
    PasswordCallback getPasswordCallback();

    void setKeystoreStream(java.io.InputStream stream);

    void setTruststoreStream(java.io.InputStream stream);

    void addSeedStream(java.io.InputStream stream);
}
```

```
package IceSSL;

public interface Plugin extends Ice.Plugin
{
    void setContext(javax.net.ssl.SSLContext context);
    javax.net.ssl.SSLContext getContext();

    void setCertificateVerifier(CertificateVerifier verifier);
    CertificateVerifier getCertificateVerifier();

    void setPasswordCallback(PasswordCallback callback);
    PasswordCallback getPasswordCallback();

    void setKeystoreStream(java.io.InputStream stream);

    void setTruststoreStream(java.io.InputStream stream);

    void addSeedStream(java.io.InputStream stream);
}
```

The methods are summarized below:

- `setContext`
  `getContext`
  These methods are for advanced use cases and rarely used in practice.
- `setCertificateVerifier`
  `getCertificateVerifier`
  These methods install and retrieve a custom certificate verifier object that the plug-in invokes for each new connection. `getCertif`

icateVerifier returns null if a verifier has not been set.

- setPasswordCallback
  getPasswordCallback
  These methods install and retrieve a password callback object that supplies IceSSL with passwords. getPasswordCallback returns null if a callback has not been set. Using setPasswordCallback is a more secure alternative to setting passwords in clear-text configuration files.
- setKeystoreStream
  Supplies an input stream for a keystore containing the key pair. The IceSSL.Keystore property is ignored if this method is called with a non-null value. You may supply the same input stream object to this method and to setTruststoreStream if your keystore contains your key pair as well as your trusted CA certificates.
- setTruststoreStream
  Supplies an input stream for a truststore containing your trusted CA certificates. The IceSSL.Truststore property is ignored if this method is called with a non-null value. You may supply the same input stream object to this method and to setKeystoreStream if your keystore contains your key pair as well as your trusted CA certificates.
- addSeedStream
  Adds an input stream that supplies seed data for the random number generator. You may call this method multiple times if necessary.

Obtaining SSL Connection Information in Java

You can obtain information about any SSL connection using the getInfo operation on a Connection object. IceSSL defines the following type in Slice:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ConnectionInfo underlying;
        bool incoming;
        string adapterName;
        string connectionId;
    }
}

module IceSSL
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string cipher;
        ["java:type:java.security.cert.Certificate[]"]
        Ice::StringSeq certs;
        bool verified;
    }
}
```

The Ice::ConnectionInfo object can be narrowed to IceSSL::ConnectionInfo for an SSL connection.

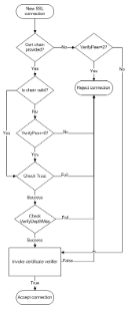
The certs member contains the peer's certificate chain; the java:type metadata changes the mapping to an array of java.security.cert.Certificate objects. The array is structured so that the first element is the peer's certificate, followed by its signing certificates in the order they appear in the chain, with the root CA certificate as the last element. The array is empty if the peer did not present a certificate chain.

The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The `verified` member indicates whether IceSSL was able to successfully verify the peer's certificate.

The inherited `underlying` data member contains the connection information of the underlying transport (if SSL is based on TCP, this member will contain an instance of `Ice::TCPEndpointInfo` which you can use to retrieve the remote and local addresses). The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). The `connectionId` data member matches the connection identifier set on the proxy. Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

Installing a Certificate Verifier in Java

A new connection undergoes a series of verification steps before an application is allowed to use it. The low-level SSL engine executes certificate validation procedures and, assuming the certificate chain is successfully validated, IceSSL performs additional verification as directed by its configuration properties. If a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to accept or reject the connection. The value of the `IceSSL.VerifyPeer` property also plays an important role here. We've summarized the process in the following flow chart:



The `CertificateVerifier` interface has only one method:

Java Java Compat

```
package com.zeroc.IceSSL;

public interface CertificateVerifier
{
    boolean verify(ConnectionInfo info);
}
```

```
package IceSSL;

public interface CertificateVerifier
{
    boolean verify(ConnectionInfo info);
}
```

IceSSL rejects the connection if `verify` returns `false`, and allows it to proceed if the method returns `true`. The `verify` method receives a `ConnectionInfo` object that describes the connection's attributes.

The following class is a simple implementation of a certificate verifier:

Java

```
import java.security.cert.X509Certificate;
import javax.security.auth.x500.X500Principal;

class Verifier implements com.zeroc.IceSSL.CertificateVerifier
{
    public boolean verify(com.zeroc.IceSSL.ConnectionInfo info)
    {
        if(info.certs != null)
        {
            X509Certificate cert = (X509Certificate)info.nativeCerts[0];
            X500Principal p = cert.getIssuerX500Principal();
            if(p.getName().toLowerCase().indexOf("zeroc") != -1)
            {
                return true;
            }
        }
        return false;
    }
}
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

Java

```
com.zeroc.IceSSL.Plugin sslPlugin = // ...
sslPlugin.setCertificateVerifier(new Verifier());
```

You should install the verifier before any SSL connections are established. An alternate way of installing the verifier is to define the `IceSSL.CertVerifier` property with the class name of your verifier implementation. IceSSL instantiates the class using its default constructor.

You can also install a certificate verifier using a custom plug-in to avoid making changes to the code of an existing application.

> The Ice run time calls the `verify` method during the connection-establishment process, therefore delays in the `verify` implementation have a direct impact on the performance of the application. Do not make remote invocations from your implementation of `verify`.

Converting Certificates in Java

Java does not provide a simple way to create a certificate object from a PEM-encoded string, therefore IceSSL offers the following convenience method:

Java Java Compat

```
package com.zeroc.IceSSL;

public final class Util
{
    public static java.security.cert.X509Certificate
createCertificate(String certPEM)
        throws java.security.cert.CertificateException;
}
```

```
package IceSSL;

public final class Util
{
    public static java.security.cert.X509Certificate
createCertificate(String certPEM)
        throws java.security.cert.CertificateException;
}
```

Given a string in the PEM format, `createCertificate` returns the equivalent `X509Certificate` object.

See Also

- Using Connections
- Public Key Infrastructure
- Configuring IceSSL
- Advanced IceSSL Topics

Programming IceSSL in .NET

This page describes the .NET API for the IceSSL plug-in.

On this page:

The IceSSL Plugin Interface in C#

Applications can interact directly with the IceSSL plug-in using the native C# interface `IceSSL.Plugin`. A reference to a `Plugin` object must be obtained from the communicator in which the plug-in is installed:

C#

```
Ice.Communicator comm = // ...
Ice.PluginManager pluginMgr = comm.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
```

The `Plugin` interface supports the following methods:

C#

```
namespace IceSSL
{
    using System.Security.Cryptography.X509Certificates;

    abstract public class Plugin : Ice.Plugin
    {
        abstract public void
        setCertificates(X509Certificate2Collection certs);

        abstract public void
        setCACertificates(X509Certificate2Collection certs);
        abstract public void
        setCertificateVerifier(CertificateVerifier verifier);

        abstract public CertificateVerifier
        getCertificateVerifier();

        abstract public void
        setPasswordCallback(PasswordCallback callback);

        abstract public PasswordCallback
        getPasswordCallback();
    }
}
```

The methods are summarized below:

- `setCertificates`
  `setCACertificates`
  These methods are for advanced use cases and rarely used in practice.
- `setCertificateVerifier`
  `getCertificateVerifier`
  These methods install and retrieve a custom certificate verifier object that the plug-in invokes for each new connection. `getCertificateVerifier` returns null if a verifier has not been set.
- `setPasswordCallback`
  `getPasswordCallback`
  These methods install and retrieve a password callback object that supplies IceSSL with passwords. `getPasswordCallback` returns null if a callback has not been set. Using `setPasswordCallback` is a more secure alternative to setting passwords in clear-text configuration files.

Obtaining SSL Connection Information in C#

You can obtain information about any SSL connection using the `getInfo` operation on a `Connection` object. IceSSL defines the following type in Slice:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ConnectionInfo underlying;
        bool incoming;
        string adapterName;
        string connectionId;
    }
}

module IceSSL
{
    local class ConnectionInfo extends Ice::IPConnectionInfo
    {
        string cipher;

["cs:type:System.Security.Cryptography.X509Certificates.X509Certificate
2[]"]
        Ice::StringSeq certs;
        bool verified;
    }
}
```

The `Ice::ConnectionInfo` object can be narrowed to `IceSSL::ConnectionInfo` for an SSL connection.

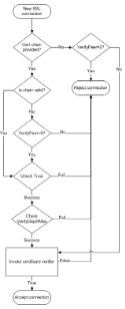
The `certs` member contains the peer's certificate chain; the `cs:type` metadata changes the mapping to an array of `X509Certificate2` objects. The array is structured so that the first element is the peer's certificate, followed by its signing certificates in the order they appear in the chain, with the root CA certificate as the last element. The array is empty if the peer did not present a certificate chain.

The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The `verified` member indicates whether IceSSL was able to successfully verify the peer's certificate.

The inherited `underlying` data member contains the connection information of the underlying transport (if SSL is based on TCP, this member will contain an instance of `Ice::TCPEndpointInfo` which you can use to retrieve the remote and local addresses). The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). The `connectionId` data member matches the connection identifier set on the proxy. Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

Installing a Certificate Verifier in C#

A new connection undergoes a series of verification steps before an application is allowed to use it. The low-level SSL engine executes certificate validation procedures and, assuming the certificate chain is successfully validated, IceSSL performs additional verification as directed by its configuration properties. If a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to accept or reject the connection. The value of the `IceSSL.VerifyPeer` property also plays an important role here. We've summarized the process in the following flow chart:



The `CertificateVerifier` interface has only one method:

C#

```
namespace IceSSL
{
    public interface CertificateVerifier
    {
        bool verify(ConnectionInfo info);
    }
}
```

IceSSL rejects the connection if `verify` returns `false`, and allows it to proceed if the method returns `true`. The `verify` method receives a `NativeConnectionInfo` object that describes the connection's attributes.

The following class is a simple implementation of a certificate verifier:

C#

```
using System.Security.Cryptography.X509Certificates;

class Verifier : IceSSL.CertificateVerifier
{
    public boolean verify(IceSSL.ConnectionInfo info)
    {
        if(info.nativeCerts != null)
        {
            X500DistinguishedName dn = info.nativeCerts[0].IssuerName;
            if(dn.Name.ToLower().Contains("zeroc"))
            {
                return true;
            }
        }
        return false;
    }
}
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

C#

```
IceSSL.Plugin sslPlugin = // ...
sslPlugin.setCertificateVerifier(new Verifier());
```

You should install the verifier before any SSL connections are established. An alternate way of installing the verifier is to define the `IceSSL.CertVerifier` property with the class name of your verifier implementation. IceSSL instantiates the class using its default constructor.

You can also install a certificate verifier using a custom plug-in to avoid making changes to the code of an existing application.

> The Ice run time calls the `verify` method during the connection-establishment process, therefore delays in the `verify` implementation have a direct impact on the performance of the application. Do not make remote invocations from your implementation of `verify`.

Converting Certificates in C#

IceSSL offers the following convenience method to create a certificate object from a PEM-encoded string:

C#

```
namespace IceSSL
{
    using System.Security.Cryptography.X509Certificates;

    public sealed class Util
    {
        // ...

        public static X509Certificate2
createCertificate(string certPEM);
    }
}
```

Given a string in the PEM format, `createCertificate` returns the equivalent `X509Certificate2` object.

See Also

- Using Connections
- Public Key Infrastructure
- Configuring IceSSL
- Advanced IceSSL Topics

Programming IceSSL in Other Languages

Although IceSSL's native plug-in API is only available to programs written in C++, .NET and Java, you can still obtain some useful information in other languages.

On this page:

Obtaining SSL Connection Information

You can obtain information about any SSL connection using the `getInfo` operation on a `Connection` object. IceSSL defines the following types in Slice:

Slice

```
module Ice
{
    local class ConnectionInfo
    {
        ConnectionInfo underlying;
        bool incoming;
        string adapterName;
        string connectionId;
    }
}

module IceSSL
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string cipher;
        Ice::StringSeq certs;
        bool verified;
    }
}
```

For an SSL connection, `getInfo` returns an instance of the subclass `IceSSL::ConnectionInfo`.

The `certs` member contains the peer's certificate chain, represented here as a sequence of strings containing the PEM-encoded certificates. The array is structured so that the first element is the peer's certificate, followed by its signing certificates in the order they appear in the chain, with the root CA certificate as the last element. The array is empty if the peer did not present a certificate chain.

The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The `verified` member indicates whether IceSSL was able to successfully verify the peer's certificate.

The inherited `underlying` data member contains the connection information of the underlying transport (if SSL is based on TCP, this member will contain an instance of `Ice::TCPEndpointInfo` which you can use to retrieve the remote and local addresses). The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). The `connectionId` data member matches the connection identifier set on the proxy. Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

Converting Certificates

The code samples below demonstrate how to convert the encoded certificates into certificate objects:

Python Ruby PHP

```
from cryptography import x509
from cryptography.hazmat.backends import default_backend
...
info = proxy.ice_getConnection().getInfo()
if proxy.ice_isSecure(): # Only for a secure proxy
    certs = []
    for s in info.certs:
        certs.append(x509.load_pem_x509_certificate(bytes(s, "utf8"),
default_backend()))
```

Python doesn't currently have any built-in modules for manipulating certificates, but the `cryptography` package offers one solution.

```
require 'openssl'
...
info = proxy.ice_getConnection().getInfo()
if proxy.ice_isSecure() # Only for a secure proxy
    certs = []
    for s in info.certs
        certs.push(OpenSSL::X509::Certificate.new(s))
    end
end
```

```
$info = $proxy->ice_getConnection()->getInfo();
if($proxy->ice_isSecure()) // Only for a secure proxy
{
    $certs = array();
    for($x = 0; $x < count($info->certs); $x++)
    {
        array_push($certs, openssl_x509_parse($info->certs[x]));
    }
}
```

See Also

- Using Connections
- Public Key Infrastructure
- Advanced IceSSL Topics

Advanced IceSSL Topics

This page discusses some additional capabilities of the IceSSL plug-in.

On this page:

Managing Certificate Passwords

IceSSL may need to obtain a password if it loads a file that contains secure data, such as an encrypted private key. An application can supply a plain-text password in a configuration property, but doing so is a potential security risk. For example, if you define the property on the application's command-line, it may be possible for other users on the same host to see the password simply by obtaining a list of active processes. If you define the property in a configuration file, the password is only as secure as the file in which it is defined.

In highly secure environments where access to a host is tightly restricted, a password can safely be supplied as a plain-text configuration property, or the need for the password can be eliminated altogether by using unsecured key files.

In situations where password security is a concern, the application generally needs to take additional action.

Dynamic Properties

A common technique is to prompt the user for a password and transfer the user's input to a configuration property that the application defines dynamically, as shown below:

C++

```
string password = // ...
Ice::InitializationData initData;
initData.properties = Ice::createProperties(argc, argv);
initData.properties->setProperty("IceSSL.Password", password);
Ice::CommunicatorHolder ich(initData);
```

The password must be present in the property set before the communicator is initialized, since IceSSL needs the password during its initialization, and the communicator initializes plug-ins automatically by default.

Password Callbacks in C++

If a password is required but the application has not configured one, IceSSL prompts the user at the terminal during the plug-in's initialization. This behavior is not suitable for some types of applications, such as a program that runs automatically at system startup as a Unix daemon or Windows service.

A terminal prompt is equally undesirable for graphical applications, which would generally prefer to prompt the user in an application window. The dynamic property technique described in the previous section is usually appropriate in this situation.

If your application must supply a password, and you do not want to use a configuration property or a terminal prompt, your remaining option is supply your own password prompt implementation to the plug-in using the `setPasswordPrompt` function:

C++11 C++98

```
namespace IceSSL
{
    class Plugin : public Ice::Plugin
    {
    public:
        ...
        virtual void setPasswordPrompt(std::function<std::string()>) =
0;
        ...
    };
}
```

```
namespace IceSSL
{
    class PasswordPrompt : public IceUtil::Shared
    {
    public:

        virtual std::string getPassword() = 0;
    };
    typedef IceUtil::Handle<PasswordPrompt> PasswordPromptPtr;

    class Plugin : public Ice::Plugin
    {
    public:
        ...
        virtual void setPasswordPrompt(const PasswordPromptPtr&) = 0;
        ...
    };
}
```

IceSSL calls the supplied function (or `getPassword` in C++98) when a password is required. If the returned password is incorrect, IceSSL tries again, up to the limit defined by the `IceSSL.PasswordRetryMax` property.

Note that you must delay the initialization of the IceSSL plug-in until after the installation of your password prompt. To illustrate this point, consider the following example:

C++11 C++98

```
shared_ptr<Ice::Communicator> communicator = // ...
auto pluginMgr = communicator->getPluginManager();
auto plugin = pluginMgr->getPlugin("IceSSL");
auto sslPlugin = dynamic_point_cast<IceSSL::Plugin>(plugin);
sslPlugin->setPasswordPrompt(newPrompt); // OOPS!
```

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr = communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin = IceSSL::PluginPtr::dynamicCast(plugin);
sslPlugin->setPasswordPrompt(new Prompt); // OOPS!
```

This code is incorrect because the new password prompt is installed too late: the communicator is already initialized, which means IceSSL has already attempted to load the file that required a password.

The correct approach is to define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

C++11 C++98

```
shared_ptr<Ice::Communicator> communicator = // ...
auto pluginMgr = communicator->getPluginManager();
auto plugin = pluginMgr->getPlugin("IceSSL");
auto sslPlugin = dynamic_pointer_cast<IceSSL::Plugin>(plugin);
sslPlugin->setPasswordPrompt(newPrompt);
pluginMgr->initializePlugins();
```

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr = communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin = IceSSL::PluginPtr::dynamicCast(plugin);
sslPlugin->setPasswordPrompt(new Prompt);
pluginMgr->initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordPrompt` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Password Callbacks in Java

If you do not want to use configuration properties to define passwords, you can install a `PasswordCallback` object in the plug-in using a configuration property, or using the `setPasswordCallback` method. The `PasswordCallback` interface has the following definition:

Java Java Compat

```
package com.zeroc.IceSSL;

public interface PasswordCallback
{
    char[] getPassword(String alias);
    char[] getTruststorePassword();
    char[] getKeystorePassword();
}
```

```
package IceSSL;

public interface PasswordCallback
{
    char[] getPassword(String alias);
    char[] getTruststorePassword();
    char[] getKeystorePassword();
}
```

The methods are described below:

- `getPassword`
  Supplies the password for the key with the given alias. The return value must not be null.
- `getTruststorePassword`
  Supplies the password for a truststore. The method may return null, in which case the integrity of the truststore is not verified.
- `getKeystorePassword` to obtain the password for a keystore.
  Supplies the password for a keystore. The method may return null, in which case the integrity of the keystore is not verified.

For each of these methods, IceSSL clears the contents of the returned array as soon as possible.

The simplest way to install the callback is by defining the configuration property `IceSSL.PasswordCallback`. The property's value is the name of your callback implementation class. IceSSL instantiates the class using its default constructor.

To install the callback manually, you must delay the initialization of the IceSSL plug-in until after the `PasswordCallback` object is installed. To illustrate this point, consider the following example:

Java

```
Communicator communicator = // ...
PluginManager pluginMgr = communicator.getPluginManager();
Plugin plugin = pluginMgr.getPlugin("IceSSL");
com.zeroc.IceSSL.Plugin sslPlugin = (com.zeroc.IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI()); // OOPS!
```

This code is incorrect because the `PasswordCallback` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to retrieve the certificate that required a password.

The correct approach is to define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

Java

```
Communicator communicator = // ...
PluginManager pluginMgr = communicator.getPluginManager();
Plugin plugin = pluginMgr.getPlugin("IceSSL");
com.zeroc.IceSSL.Plugin sslPlugin = (com.zeroc.IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI());
pluginMgr.initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordCallback` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Password Callbacks in .NET

If you do not want to use configuration properties to define passwords, you can install a `PasswordCallback` object in the plug-in using a configuration property, or using the `setPasswordCallback` method. The `PasswordCallback` interface has the following definition:

C#

```
using System.Security;

public interface PasswordCallback
{
    SecureString getPassword(string file);
    SecureString getImportPassword(string file);
}
```

The methods are described below:

- `getPassword`
  Supplies the password for the given file. The method may return null if no password is required.
- `getImportPassword`
  Supplies the password for a file from which certificates are imported into the certificate store. The method may return null if no password is required.

The simplest way to install the callback is by defining the configuration property `IceSSL.PasswordCallback`. The property's value is the name of your callback implementation class. IceSSL instantiates the class using its default constructor.

To install the callback manually, you must delay the initialization of the IceSSL plug-in until after the `PasswordCallback` object is installed. To illustrate this point, consider the following example:

C#

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI()); // OOPS!
```

This code is incorrect because the `PasswordCallback` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to retrieve the certificate that required a password.

The correct approach is to define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

C#

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI());
pluginMgr.initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordCallback` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Manually Configuring IceSSL

The `Plugin` interface supports a method in each of the supported language mappings that provides an application with more control over the plug-in's configuration.

In C++ with OpenSSL and in Java, an application can call the `setContext` method to supply a pre-configured "context" object used by the underlying SSL engines. In .NET, the `setCertificates` method accepts a collection of certificates that the plug-in should use. In all cases, using one of these methods causes IceSSL to ignore (at a minimum) the configuration properties related to certificates and keys. The application is responsible for accumulating its certificates and keys, and must also deal with any password requirements.

Describing the use of these plug-in methods in detail is outside the scope of this manual, however it is important to understand their prerequisites. In particular, the application needs to have the communicator load the plug-in but not actually initialize it until after the application has had a chance to interact directly with it. (The previous section showed examples of this technique.) The application must define the `Ice.InitPlugins` configuration property:

```
Ice.InitPlugins=0
```

With this setting, the application becomes responsible for completing the plug-in initialization process by invoking `initializePlugins` on the `PluginManager`. The C# example below demonstrates the proper steps:

C#

```
Ice.Communicator comm = // ...
Ice.PluginManager pluginMgr = comm.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
X509Certificate2Collection certs = // ...
sslPlugin.setCertificates(certs);
pluginMgr.initializePlugins();
```

Using Custom Plug-ins with IceSSL

The Ice plug-in facility is not restricted to protocol implementations. Ice only requires that a plug-in implement the `Ice::Plugin` interface and support the language-specific mechanism for dynamic loading.

The customization options of the IceSSL plug-in make it possible for you to install an application-specific implementation of a certificate verifier in an existing program. For example, you could install a custom certificate verifier in a Glacier2 router without the need to modify Glacier2's source code or rebuild the executable. You would have to write a C++ plug-in to accomplish this, since Glacier2 is written in C++. In short, your plug-in must interact with the IceSSL plug-in and install a certificate verifier.

For this technique to work, it is important that the plug-ins be loaded in a particular order. Specifically, the IceSSL plug-in must be loaded first, followed by the certificate verifier plug-in. By default, Ice loads plug-ins in an undefined order, but you can use the property `Ice.PluginLoadOrder` to specify a particular order.

As an example, let's write a plug-in that installs our simple C++ certificate verifier. Here is the definition of our plug-in class:

C++11 C++98

```
class VerifierPlugin : public Ice::Plugin
{
public:
    VerifierPlugin(const std::shared_ptr<Ice::Communicator>& communicat
or)
        _communicator(communicator)
    {
    }

    virtual void initialize() override
    {
        auto pluginMgr = _communicator->getPluginManager();
        auto plugin = pluginMgr->getPlugin("IceSSL");
        auto sslPlugin =
std::dynamic_pointer_cast<IceSSL::Plugin>(plugin);

        auto verifier = [](const shared_ptr<IceSSL::ConnectionInfo>&
info)
        {
           ....
        };
        sslPlugin->setCertificateVerifier(verifier);
    }

    virtual void destroy() override
    {
    }

private:
    std::shared_ptr<Ice::Communicator> _communicator;
};
```

```
class VerifierPlugin : public Ice::Plugin
{
public:
    VerifierPlugin(const Ice::CommunicatorPtr& communicator) :
        _communicator(communicator)
    {
    }

    virtual void initialize()
    {
        Ice::PluginManagerPtr pluginMgr =
_communicator->getPluginManager();
        Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
        IceSSL::PluginPtr sslPlugin =
IceSSL::PluginPtr::dynamicCast(plugin);
        sslPlugin->setCertificateVerifier(new Verifier);
    }

    virtual void destroy()
    {
    }

private:
    Ice::CommunicatorPtr _communicator;
};
```

The class implements the two operations in the `Plugin` interface, `initialize` and `destroy`. The code in `initialize` installs the certificate verifier function or object, while nothing needs to be done in `destroy`.

The next step is to write the plug-in's factory function, which the communicator invokes to obtain an instance of the plug-in:

C++11 C++98

```
extern "C"
{
    Ice::Plugin*

createVerifierPlugin(const shared_ptr<Ice::Communicator>& communicator,
                         const string& name,
                         const Ice::StringSeq& args)
    {
        return new VerifierPlugin(communicator);
    }
}
```

```
extern "C"
{
    Ice::Plugin*
    createVerifierPlugin(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const Ice::StringSeq& args)
    {
        return new VerifierPlugin(communicator);
    }
}
```

We can give the function any name; in this example, we chose `createVerifierPlugin`.

Finally, to install the plug-in we need to define the following properties:

```
Ice.PluginLoadOrder=IceSSL,Verifier
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.Verifier=Verifier:createVerifierPlugin
```

The value of `Ice.PluginLoadOrder` guarantees that IceSSL is loaded first. The plug-in specification `Verifier:createVerifierPlugin` identifies the name of the shared library or DLL and the name of the registration function.

There are a few more details you must attend to, such as ensuring that the factory function is exported properly and building the shared library or DLL that contains the new plug-in. Our discussion of the plug-in facility provides more information on developing a plug-in.

See Also

- Service Helper Class
- Configuring IceSSL
- Programming IceSSL in C++
- Programming IceSSL in Java
- Programming IceSSL in .NET
- Plug-in Facility
- IceSSL.*
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

Setting up a Certificate Authority

During development, it is convenient to have a simple way of creating new certificates. OpenSSL includes all of the necessary infrastructure for setting up your own certificate authority (CA), but it requires getting more familiar with OpenSSL than is really necessary. To simplify the process, the zeroc-icecertutils PyPi package provides a Python library and and an `iceca` script that allows you to quickly perform the essential tasks:

- initializing a new root CA
- generating new certificates
- converting certificates to match platform-specific requirements.

You are not obligated to use this package; IceSSL accepts certificates from any source as long as they are provided in the appropriate formats. However, you may find this tool sufficient for your development needs, and possibly even for your deployed application as well.

You will find more information on the `IceCertUtils` package and the `iceca` utility on the zeroc-icecertutils PyPI page. You can also take a look at the `makecerts.py` scripts in the `certs` and `cpp/IceGrid/secure` directories from the Ice demos repository for examples on how to use the `IceCertUtils` Python library.

See Also

- Configuring IceSSL

IceStringConverter

The Ice for C++ library includes the `IceStringConverter` plug-in, which is described in the language mapping chapters:

- The C++11 Ice String Converter Plug-in
- The C++98 Ice String Converter Plug-in

Using Plugins with Static Libraries

When a plug-in is packaged in a static library, you need to link your application with the plug-in's static library and call a `register` function from your application to ensure the corresponding code gets correctly included.

Ice provides the following plug-in registration functions for C++ and Objective-C applications:

C++ Objective-C

C++

```
namespace Ice
{
    void registerIceUDP(bool loadOnInitialize = true);
    void registerIceIAP(bool loadOnInitialize = true);
    void registerIceBT(bool loadOnInitialize = true);
    void registerIceDiscovery(bool loadOnInitialize = true);
    void registerIceLocatorDiscovery(bool loadOnInitialize = true);
    void registerIceSSL(bool loadOnInitialize = true);
    void registerIceWS(bool loadOnInitialize = true);
    void registerIceStringConverter(bool loadOnInitialize = true);
}
```

Objective-C

```
void ICEregisterIceUDP(BOOL loadOnInitialize);
void ICEregisterIceIAP(BOOL loadOnInitialize);
void ICEregisterIceDiscovery(BOOL loadOnInitialize);
void ICEregisterIceLocatorDiscovery(BOOL loadOnInitialize);
void ICEregisterIceSSL(BOOL loadOnInitialize);
void ICEregisterIceWS(BOOL loadOnInitialize);
void ICEregisterIceStringConverter(BOOL loadOnInitialize = true);
```

For example, if you are using a static IceSSL plug-in, link your application with `libIceSSL.a` and call `Ice::registerIceSSL` from your application. The `UDP`, `WS`, and `StringConverter` plug-ins are in the main Ice library (`libIce.a`).

When the bool parameter is `true` (the default in C++), the plug-in is always loaded (created) during communicator initialization, even if `Ice.Plugin.`*`name`* is not set. When `false`, the plug-in is loaded (created) during communication initialization only if `Ice.Plugin.`*`name`* is set to a non-empty value (e.g., `Ice.Plugin.IceSSL=1`).

`registerIceIAP` is currently available only on iOS.

The table below summarizes the available registration functions and the static libraries where they are defined:

C++ Objective-C

Plug-in Name	Functionality	Function	Library
IceUDP	udp transport	`Ice::registerIceUDP`	`libIce.a`
IceWS	ws and wss transports	`Ice::registerIceWS`	`libIce.a`

IceStringConverter	convert native string encoding to/from UTF-8	`Ice::registerIceStringConverter`	`libIce.a`
IceIAP	`iap` and `iaps` transports	`Ice::registerIceIAP`	`libIceIAP.a`
IceBT	`bt` and `bts` transports	`Ice::registerIceBT`	`libIceBT.a`
IceSSL	`icessl` transport	`Ice::registerIceSSL`	`libIceSSL.a`
IceDiscovery	Location service implemented using UDP multicast	`Ice::registerIceDiscovery`	`libIceDiscovery.a`
IceLocatorDiscovery	Location service implemented using UDP multicast	`Ice::registerIceLocatorDiscovery`	`libIceLocatorDiscovery.a`

Plug-in Name	Functionality	Function	Library
IceUDP	`udp` transport	`ICEregisterIceUDP`	`libIce.a`
IceWS	`ws` and `wss` transports	`ICEregisterIceWS`	`libIce.a`
IceStringConverter	convert native string encoding to/from UTF-8	`ICEregisterIceStringConverter`	`libIce.a`
IceIAP	`iap` and `iaps` transports	`ICEregisterIceIAP`	`libIceIAP.a`
IceSSL	`icessl` transport	`ICEregisterIceSSL`	`libIceSSL.a`
IceDiscovery	Location service implemented using UDP multicast	`ICEregisterIceDiscovery`	`libIceDiscovery.a`
IceLocatorDiscovery	Location service implemented using UDP multicast	`ICEregisterIceLocatorDiscovery`	`libIceLocatorDiscovery.a`

If you are developing your own plug-in, you will also need to register your plug-in factory function using `Ice::registerPluginFactory`. Refer to the Plug-in API for more information on this function.

Ice Services

This section describes the services included with Ice.

Topics

Glacier2

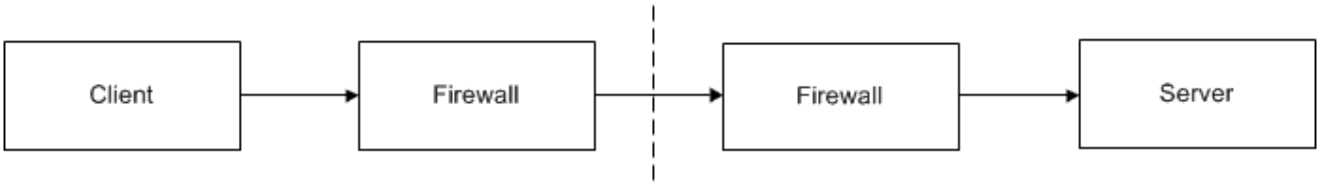
Glacier2 is a lightweight firewall traversal solution for Ice applications.

> We present many examples of client/server applications in this manual, most of which assume that the client and server programs are running either on the same host, or on multiple hosts with no network restrictions. We can justify this assumption because this is an instructional text, but a real-world network environment is usually much more complicated: client and server hosts with access to public networks often reside behind protective router-firewalls that not only restrict incoming connections, but also allow the protected networks to run in a private address space using Network Address Translation (NAT). These features, which are practically mandatory in today's hostile network environments, also disrupt the ideal world in which our examples are running.

Topics

Common Firewall Traversal Issues

Let's assume that a client and server need to communicate over an untrusted network, and that the client and server hosts reside in private networks behind firewalls:

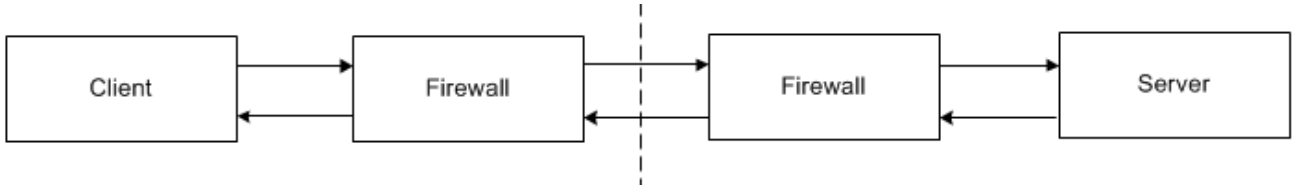


Scenario 1: Client request in a typical network.

Although the diagram looks fairly straightforward, there are several troublesome issues:

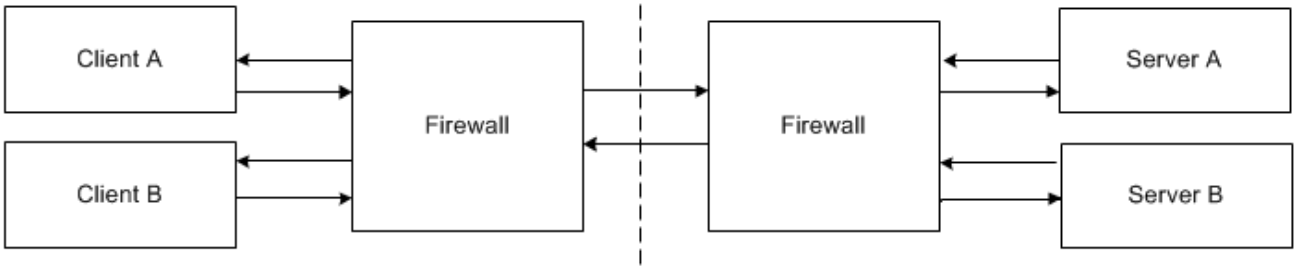
- A dedicated port on the server's firewall must be opened and configured to forward messages to the server.
- If the server uses multiple endpoints (e.g., to support both TCP and SSL), then a firewall port must be dedicated to each endpoint.
- The client's proxy must be configured to use the server's "public" endpoint, which is the host name and dedicated port of the firewall.
- If the server returns a proxy as the result of a request, the proxy must not contain the server's private endpoint because that endpoint is inaccessible to the client.

To complicate the scenario even further, the illustration below adds a callback from the server to the client. Callbacks imply that the client is also a server, therefore all of the issues associated with previous illustration now apply to the client as well.



Scenario 2: Callbacks in a typical network.

As if this was not complicated enough already, the illustration below adds multiple clients and servers. Each additional server (including clients requiring callbacks) adds more work for the firewall administrator as more ports are dedicated to forwarding requests.



Scenario 3: Multiple clients and servers with callbacks in a typical network.

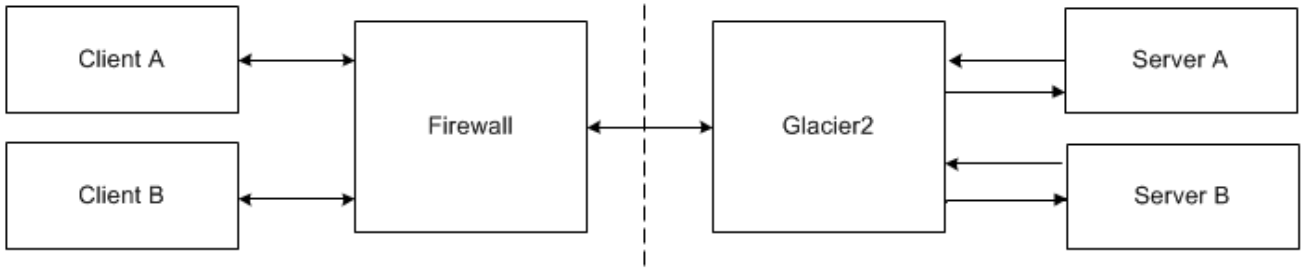
Clearly, these scenarios do not scale well, and are unnecessarily complex. Fortunately, Ice provides a solution in Glacier2.

See Also

- About Glacier2
- How Glacier2 Works
- Getting Started with Glacier2

About Glacier2

Glacier2, the router-firewall for Ice applications, addresses common firewall traversal issues with minimal impact on clients or servers (or firewall administrators). In the illustration below, Glacier2 becomes the server firewall for Ice applications. What is not obvious in the diagram, however, is how Glacier2 eliminates much of the complexity of firewall traversal.



Complex network environments are a fact of life. Unfortunately, the cost of securing an enterprise's network is increased application complexity and administrative overhead. Glacier2 helps to minimize these costs by providing a low-impact, efficient and secure router for Ice applications.

Glacier2 has the following advantages and limitations.

Advantages

- Clients often require only minimal changes to use Glacier2.
- Only one front-end port is necessary to support any number of servers, allowing a Glacier2 router to easily receive connections from a port-forwarding firewall.
- The number of connections to back-end servers is reduced. Glacier2 effectively acts as a connection concentrator, establishing a single connection to each back-end server to forward requests from any number of clients. Similarly, connections from back-end servers to Glacier2 for the purposes of sending callbacks are also concentrated.
- Servers are unaware of Glacier2's presence, and require no modifications whatsoever to use Glacier2. From a server's perspective, Glacier2 is just another local client, therefore servers are no longer required to advertise "public" endpoints in the proxies they create. Furthermore, back-end services such as IceGrid can continue to be used transparently via a Glacier2 router.
- Callbacks through Glacier2 are supported without requiring new connections from servers to clients. In other words, a callback from a server to a client is sent over an existing connection from the client to the server, thereby eliminating the administrative requirements associated with supporting callbacks in the client firewall.
- Glacier2 requires no knowledge of the application's Slice definitions and therefore is very efficient: it routes request and reply messages without unmarshalling the message contents.
- In addition to its primary responsibility of forwarding Ice requests, Glacier2 offers support for user-defined session management and authentication, inactivity timeouts, and request buffering and batching.

Limitations

- Datagram protocols, such as UDP, are not supported.
- Callback objects in a client must use a Glacier2-supplied category in their identities.

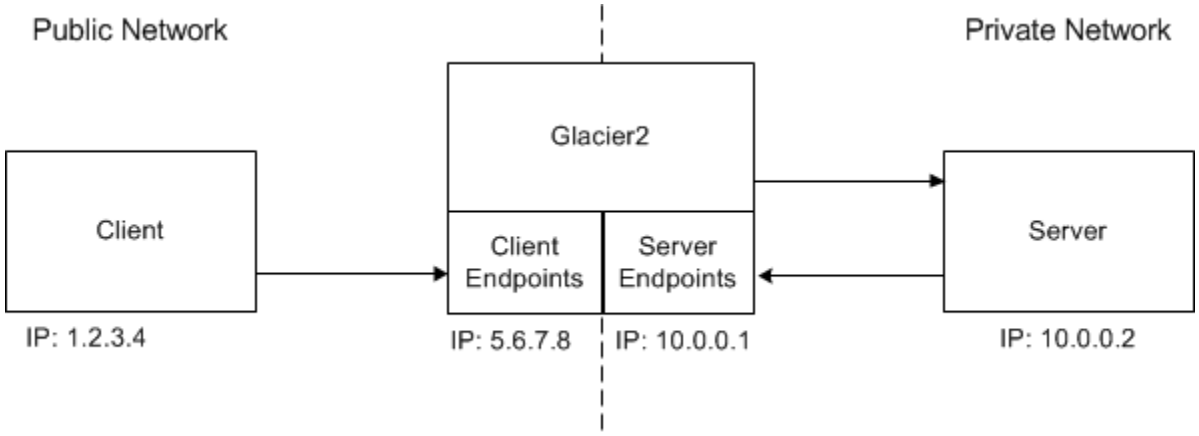
See Also

- How Glacier2 Works
- Common Firewall Traversal Issues
- Callbacks through Glacier2
- IceGrid

How Glacier2 Works

The Ice core supports a generic router facility, represented by the `Ice::Router` interface, that allows a third-party service to "intercept" requests on a properly-configured proxy and deliver them to the intended server. Glacier2 is an implementation of this service, although other implementations are certainly possible.

Glacier2 normally runs on a host in the private network behind a port-forwarding firewall, but it can also operate on a host with access to both public and private networks. In this configuration it follows that Glacier2 must have endpoints on each network.



> For the sake of example, the router's public address is 5.6.7.8 and its private address is 10.0.0.1.

In the client, proxies must be configured to use Glacier2 as a router. This configuration can be done statically for all proxies created by a communicator, or programmatically for a particular proxy. A proxy configured to use a router is called a *routed proxy*.

When a client invokes an operation on a routed proxy, the client connects to one of Glacier2's client endpoints and sends the request as if Glacier2 is the server. Glacier2 then establishes an outgoing connection to the client's intended server in the private network, forwards the request to that server, and returns the reply (if any) to the client. Glacier2 is essentially acting as a local client on behalf of the remote client.

If a server returns a proxy as the result of an operation, that proxy contains the server's endpoints in the private network, as usual. (Remember, the server is unaware of Glacier2's presence, and therefore assumes that the proxy is usable by the client that requested it.) In the absence of a router, a client would receive an exception if it attempted to use such a proxy. When configured with a router, however, the client ignores the proxy's endpoints and always sends requests to the router's client endpoints instead.

Glacier2's server endpoints, which reside in the private network, are only used when a server makes a callback to a client.

See Also

- Callbacks through Glacier2
- Configuring Glacier2 behind an External Firewall

Getting Started with Glacier2

On this page:

Using Glacier2

Using Glacier2 in a minimal configuration involves the following tasks:

1. Write a configuration file for the router.
2. Write a password file for the router. (Glacier2 also supports other ways to authenticate users.)
3. Decide whether to use the router's internal session manager, or supply your own session manager.
4. Start the router on a host with access to the public and private networks.
5. Modify the client configuration to use the router.
6. Modify the client to create a router session.
7. Ensure that the router session remains active for as long as the client requires it.

> For the sake of example, the router's public address is 5.6.7.8 and its private address is 10.0.0.1.

Configuring the Router

The following router configuration properties establish the necessary endpoint and define when a session expires due to inactivity:

```
Glacier2.Client.Endpoints=tcp -h 5.6.7.8 -p 4063
Glacier2.SessionTimeout=60
```

The endpoint defined by `Glacier2.Client.Endpoints` is used by the Ice run time in a client to interact directly with the router. It is also the endpoint where requests from routed proxies are sent. This endpoint is defined on the public network interface because it must be accessible to clients. Furthermore, the endpoint uses a fixed port because clients may be statically configured with a proxy for this endpoint. The port numbers 4063 (for TCP) and 4064 (for SSL) are reserved for Glacier2 by the Internet Assigned Numbers Authority (IANA).

> This sample configuration uses TCP as the endpoint protocol, although in most cases, SSL is preferable.

A client must create a session in order to use a Glacier2 router. Our setting for the `Glacier2.SessionTimeout` property causes the router to destroy sessions that have been idle for at least 60 seconds. It is not mandatory to define a timeout, but it is recommended, otherwise session state might accumulate in the router.

Note that this configuration enables the router to forward requests from clients to servers. Additional configuration is necessary to support callbacks from servers to clients.

You must also decide which authentication scheme (or schemes) to use. A file-based mechanism is available, as are more sophisticated strategies.

If clients access a location service via the router, additional router configuration is typically necessary.

Writing a Password File

The router's simplest authentication mechanism uses an access control list in a text file consisting of username and password pairs. Passwords are encoded using the modular crypt format (MCF).

The general structure of a MCF encoded password hash is: $*identifier*$*content*, where identifier denotes the scheme used for hashing, and content denotes its contents. Glacier2 supports two types of MCF encoded password hashes:

On Windows and macOS:

- PBKDF2 using SHA-1, SHA-256, or SHA-512 as the digest algorithm.

  > PBKDF2 does not have a standard form in the MCF specification. In this case Glacier2 uses the same format as passlib.
  >
  > - $pbkdf2-digest$rounds$salt$ for SHA-256 and SHA-512.
  > - $pbkdf2$rounds$salt$ for SHA-1.

On Linux:

- Crypt using SHA-256, or SHA-512 as the digest algorithm.

The property Glacier2.CryptPasswords specifies the name of the password file:

```
Glacier2.CryptPasswords=passwords
```

The format of the password file is very simple. Each user name-password pair must reside on a separate line, with whitespace separating the user name from the password. For example, the following password file contains an entry for the user name test:

```
test
$5$rounds=110000$5rM9XIDChkgEu.S3$ov7yip4NOi1wymAZmamEv1uKPQRB0WzasoJsW
MpRT19
```

icehashpassword Helper Script

You can use the icehashpassword helper script to generate these username-password pairs. This script requires Python and pip to be installed. To install this script run:

```
> pip install zeroc-icehashpassword
```

You can now use the command icehashpassword:

```
> icehashpassword
Password:
$5$rounds=110000$5rM9XIDChkgEu.S3$ov7yip4NOi1wymAZmamEv1uKPQRB0WzasoJsW
MpRT19
```

You may also specify several optional parameters:

- -d MESSAGE_DIGEST_ALGORITHM, --digest=MESSAGE_DIGEST_ALGORITHM
- -s SALT_SIZE, --salt=SALT_SIZE
- -r ROUNDS, --rounds=ROUNDS

For example,

```
> python icehashpassword.py -r 25000 -s 32 -d sha256
Password:
$pbkdf2-sha256$25000$pJSSEuKcs9aaE.J8711LyRlD6H0P4Tyn9J7znpNyLsU$Yx7NNL
DfwwLeMbZV84X2rBYBnvrXvK/TDIQiIGabQIM
```

Note that `icehashpassword` generates PBKDF2 hashes on Windows and macOS, and Crypt hashes on Linux.

> This authentication scheme is intended for use in simple applications with a few users. Most applications should install their own custom permissions verifier.

Starting the Router

The router supports the following command-line options:

```
$ glacier2router -h
Usage: glacier2router [options]
Options:
-h, --help     Show this message.
-v, --version  Display the Ice version.
--nowarn       Suppress warnings.
```

The `--nowarn` option prevents the router from displaying warning messages at startup when it is unable to contact a permissions verifier object or a session manager object specified by its configuration.

Additional command line options are supported, including those that allow the router to run as a Windows service or Unix daemon, and Ice includes a utility to help you install the router as a Windows service.

Assuming our configuration properties are stored in a file named `config`, you can start the router with the following command:

```
$ glacier2router --Ice.Config=config
```

Configuring a Glacier2 Client

The following properties configure a client to use a Glacier2 router:

```
Ice.Default.Router=Glacier2/router:tcp -h 5.6.7.8 -p 4063
Ice.RetryIntervals=-1
```

The `Ice.Default.Router` property defines the router proxy. Its endpoints must match those in `Glacier2.Client.Endpoints`.

Setting `Ice.RetryIntervals` to -1 disables automatic retries, which are not useful for proxies configured to use a Glacier2 router.

Glacier2 Object Identities

A Glacier2 router hosts one well-known object. The default identity of this object is `Glacier2/router`, corresponding to the `Glacier2::Router` interface. If an application requires the use of multiple different (that is, not replicated) routers, it is a good idea to assign a unique identity to this object by configuring the routers with different values of the `Glacier2.InstanceName` property, as shown in the following example:

```
Glacier2.InstanceName=PublicRouter
```

This property changes the category of the object identity, which becomes `PublicRouter/router`. The client's configuration must also be changed to reflect the new identity:

```
Ice.Default.Router=PublicRouter/router:tcp -h 5.6.7.8 -p 4063
```

One exception to this rule is if you deploy multiple Glacier2 routers as replicas, for example, to gain redundancy or to distribute the message-forwarding load over a number of machines. In that case, all the routers must use the same instance name, and the router clients can use proxies with multiple endpoints, such as:

```
Ice.Default.Router=PublicRouter/router:tcp -h 5.6.7.8 -p 4063:tcp -h
6.10.7.8 -p 4063
```

A client can discover a router's proxy at run time using the `RouterFinder` interface.

Creating a Glacier2 Session

Session management is provided by the `Glacier2::Router` interface:

Slice

```
module Glacier2
{
    exception PermissionDeniedException
    {
        string reason;
    }

    interface Router extends Ice::Router
    {
        Session* createSession(string userId, string password)
            throws PermissionDeniedException,
                   CannotCreateSessionException;

        Session* createSessionFromSecureConnection()
            throws PermissionDeniedException,
                   CannotCreateSessionException;

        idempotent string getCategoryForClient();

        void refreshSession()
            throws SessionNotExistException;

        void destroySession()
            throws SessionNotExistException;

        idempotent long getSessionTimeout();

        idempotent int getACMTimeout();
    }
}
```

The interface defines two operations for creating sessions: `createSession` and `createSessionFromSecureConnection`. The router requires each client to create a session using one of these operations; only after the session is created will the router forward requests on behalf of the client.

The `createSession` operation expects a user name and password and, depending on the router's configuration, returns either a `Session` proxy or nil. When using the default authentication scheme, the given user name and password must match an entry in the router's password file in order to successfully create a session.

The `createSessionFromSecureConnection` operation does not require a user name and password because it authenticates the client using the credentials associated with the client's SSL connection to the router.

To create a session, the client typically obtains the router proxy from the communicator, downcasts the proxy to the `Glacier2::Router` interface, and invokes one of the `create` operations. The sample code below demonstrates how to do it in C++; the code will look very similar in the other language mappings.

C++11 C++98

```
auto defaultRouter = communicator->getDefaultRouter();
auto router = Ice::checkedCast<Glacier2::RouterPrx>(defaultRouter);
string username = ...;
string password = ...;
shared_ptr<Glacier2::SessionPrx> session;
try
{
    session = router->createSession(username, password);
}
catch(const Glacier2::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
catch(const Glacier2::CannotCreateSessionException& ex)
{
    cout << "cannot create session:\n" << ex.reason << endl;
}
```

```
Ice::RouterPrx defaultRouter = communicator->getDefaultRouter();
Glacier2::RouterPrx router =
Glacier2::RouterPrx::checkedCast(defaultRouter);
string username = ...;
string password = ...;
Glacier2::SessionPrx session;
try
{
    session = router->createSession(username, password);
}
catch(const Glacier2::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
catch(const Glacier2::CannotCreateSessionException& ex)
{
    cout << "cannot create session:\n" << ex.reason << endl;
}
```

If the router is configured with a session manager, the `createSession` and `createSessionFromSecureConnection` operations may return a proxy for an object implementing the `Glacier2::Session` interface (or an application-specific derived interface). The client receives a null proxy if no session manager is configured.

A non-nil session proxy returned by a `create` operation must be configured with the router that created it because the session object is only accessible via the router. If the router is configured as the client's default router at the time `createSession` or `createSessionFromSecureConnection` is invoked, as is the case in the example above, then the session proxy is already properly configured and nothing else is required. Otherwise, the client must explicitly configure the session proxy with a router using the `ice_router` proxy method.

If the client wishes to destroy the session explicitly, it must invoke `destroySession` on the router proxy. If a client does not destroy its session, the router destroys it automatically when it expires due to inactivity. A client can obtain the inactivity timeout value by calling `getSe`

ssionTimeout and keep the session alive by periodically calling refreshSession if necessary. An easier solution for keeping the session alive is to call getACMTimeout and use this value to configure the Active Connection Management behavior of the client's connection to the router:

C++11 C++98

```
int acmTimeout = router->getACMTimeout();
if(acmTimeout > 0)
{
    auto conn = router->ice_getCachedConnection();
    conn->setACM(acmTimeout, Ice::nullopt,
Ice::ACMHeartbeat::HeartbeatAlways);
}
```

```
int acmTimeout = router->getACMTimeout();
if(acmTimeout > 0)
{
    Ice::ConnectionPtr conn = router->ice_getCachedConnection();
    conn->setACM(acmTimeout, IceUtil::None, Ice::HeartbeatAlways);
}
```

The client calls getACMTimeout to retrieve the router's server-side ACM timeout and passes this value to an invocation of setACM on its connection to the router, thereby ensuring the client and router are using a consistent setting. The client also enables automatic heartbeats so that the connection remains active and prevents the router's server-side ACM from closing the connection.

Alternatively, you could configure the client's ACM with the properties Ice.ACM.Timeout and Ice.ACM.Heartbeat. Note however that these properties affect all connections created by a client. In general, we recommend configuring the connection directly as shown above.

Whether or not you use refreshSession or heartbeats to keep the session alive, Glacier2 will invoke ice_ping on the session object for each refreshSession call or heartbeat received. This ice_ping check ensures the session is still alive in the session manager. If it fails, Glacier2 destroys the session and the client is notified by the closure of the connection associated to the session.

The getCategoryForClient operation is used to implement callbacks over bidirectional connections.

Example
An example of a Glacier2 client is provided in the directory demo/Glacier2/callback.

Glacier2 Session Expiration

A Glacier2 router may be configured to destroy sessions after a period of inactivity. This feature allows the router, as well as a custom session manager, to reclaim resources acquired during the session, but it requires some coordination between the router and its clients.

Ideally you would select a session timeout that is long enough to accommodate the usage patterns of your clients. For example, a session timeout of thirty seconds is a reasonable choice for a client that invokes an operation on a back-end server once every five seconds. However, that timeout could disrupt a different client that has long periods of inactivity, such as when its invocations are prompted by human interaction.

If you cannot predict with certainty the usage patterns of your clients, we recommend configuring the clients so that they actively prevent their sessions from expiring. The simplest solution is to enable the heartbeat feature of Active Connection Management, wherein the Ice run time automatically sends a heartbeat message at regular intervals. The Glacier2 router interprets a heartbeat message as an indication that the client is still active and wishes its session to remain alive. ACM settings can be configured for all connections created by a communicator, or individually for a particular connection. Use care to ensure the client's ACM timeout and the router's session timeout are compatible.

Ice includes helper classes that simplify the task of creating a session.

Note that if a session times out, the next client invocation raises `ConnectionLostException`. To re-establish the session, the client must explicitly re-create it. If the client uses callbacks, it must also re-create the callback adapter and re-register its callback servants.

Glacier2 Session Destruction

A router session is destroyed automatically when the session expires, and when a client explicitly destroys its session. The router also destroys a session if certain connection errors occur while attempting to route a request. These errors are represented by the run-time exceptions `SocketException`, `TimeoutException`, and `ProtocolException`. In other words, if any of these exceptions occur while Glacier2 attempts to establish a connection to the target back-end server, or forward a request to the target back-end server, the router automatically destroys the session.

See Also

- Callbacks through Glacier2
- Securing a Glacier2 Router
- Glacier2 Session Management
- Glacier2.*
- Windows Services
- Active Connection Management
- Automatic Retries
- Glacier2 Application Class

Callbacks through Glacier2

Callbacks from servers to clients are commonly used in distributed applications, often to notify the client about an event such as the completion of a long-running calculation or a change to a database record. Unfortunately, supporting callbacks in a complicated network environment presents its own set of problems. Ice overcomes these obstacles using a Glacier2 router and bidirectional connections.

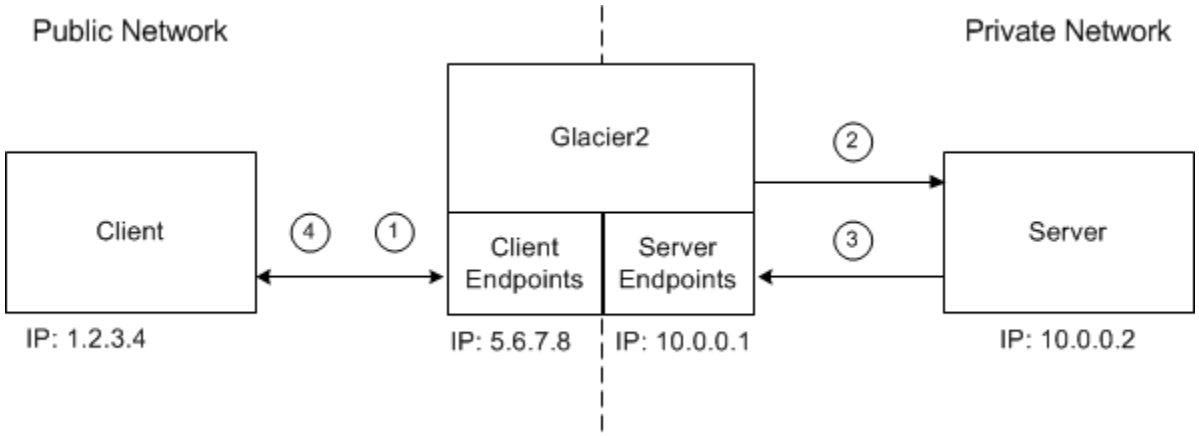
On this page:

> **Example**
> The `Glacier2/callback` demo illustrates the use of callbacks with Glacier2. The `README.md` file in the directory provides instructions on running the example, and comments in the configuration file describe the properties in detail.

Bidirectional Connections with Glacier2

While a regular unrouted connection allows requests to flow in only one direction (from client to server), a bidirectional connection enables requests to flow in both directions. This capability is necessary to circumvent the network restrictions that commonly cause firewall traversal issues, namely, client-side firewalls that prevent a server from establishing an independent connection directly to the client. By sending callback requests over the existing connection from the client to the server (more accurately, from the client to the router), we have created a virtual connection back to the client.

This diagram shows the steps involved in making a callback using Glacier2:



1. The client has a routed proxy for the server and makes an invocation. A connection is established to the router's client endpoint and the request is sent to the router.
2. The router, using information from the client's proxy, establishes a connection to the server and forwards the request. In this example, one of the arguments in the request is a proxy for a callback object in the client.
3. The server makes a callback to the client. For this to succeed, the proxy for the callback object must contain endpoints that are accessible to the server. The only path back to the client is through the router, therefore the proxy contains the router's server endpoints. The server connects to the router and sends the request.
4. The router forwards the callback request to the client using the bidirectional connection established in step 1.

The arrows in the above illustration indicate the flow of requests; notice that two connections are used between the router and the server. Since the server is unaware of the router, it does not use routed proxies, and therefore does not use bidirectional connections.

> It is also possible for applications to manually configure bidirectional connections without the use of a router.

Callbacks and Connection Closure

When a client terminates, it closes its connection to the router. If a server later attempts to make a callback to the client, the attempt fails because the router has no connection to the client over which to forward the request. This situation is no worse than if the server attempted to contact the client directly, which would be prevented by the client's firewall. However, this illustrates the inherent limitation of bidirectional connections: the lifetime of a client's callback proxy is bounded by the lifetime of the client's router session.

Configuring the Router for Callbacks

In order for the router to support callbacks from servers, it needs to have endpoints in the private network.

The configuration file shown below adds the property `Glacier2.Server.Endpoints`:

```
Glacier2.Client.Endpoints=tcp -h 5.6.7.8 -p 4063
Glacier2.Server.Endpoints=tcp -h 10.0.0.1
```

As this example shows, the server endpoint does not require a fixed port.

> Glacier2's implementation of `Ice::Router`'s `getServerProxy` returns the published endpoints of this `Glacier2.Server` object adapter.

Configuring the Client's Object Adapter with a Router

A client that receives callbacks is also a server, and therefore must have an object adapter. Typically, an object adapter has endpoints in the local network, but those endpoints are of no use to a server in our restricted network environment. We really want the client's callback proxy to contain the router's server endpoints, and we accomplish that by configuring the client's object adapter with a proxy for the router.

> Note that multiple object adapters created by the same communicator cannot use the same router.

We supply the router's proxy by creating the object adapter with `createObjectAdapterWithRouter`, or by defining the object adapter property *`adapter`*`.Router` as shown below:

```
CallbackAdapter.Router=Glacier2/router:tcp -h 5.6.7.8 -p 4063
```

For each object adapter, the Ice run time maintains a list of endpoints that are embedded in proxies created by that adapter. Normally, this list simply contains the local endpoints defined for the object adapter but, when the adapter is configured with a router, the list only contains the router's server endpoints.

An object adapter configured in this way allows the client to receive callback requests via the router. If the client also wants to service requests via local (non-routed) endpoints, the client must create a separate adapter for these requests.

> An object adapter configured with a router receives only requests over the (bidirectional) connection to the router, and collocated dispatches.

Callback Object Identities

Glacier2 assigns a unique category to each client for use in the identities of the client's callback objects. The client creates proxies that contain this identity category and pass these proxies to back-end servers for use in making callback requests to the client. This category

serves two purposes:

1. Upon receipt of a callback request from a back-end server, the router uses the request's category to identify the intended client.
2. The category is sufficiently random that, without knowing the category in advance, it is practically impossible for a misbehaving or malicious back-end server to send callback requests to an arbitrary client.

A client can obtain its assigned category by calling `getCategoryForClient` on the `Router` interface as shown in the example below:

C++11 C++98

```
std::shared_ptr<Glacier2::RouterPrx> router = // ...
string category = router->getCategoryForClient();
```

```
Glacier2::RouterPrx router = // ...
string category = router->getCategoryForClient();
```

Nested Invocations with a Router

If a router client intends to receive callbacks and make nested twoway invocations, it is important that the client be configured correctly. Specifically, you must increase the size of the client thread pool to at least two threads.

Handling Session Timeouts

If the client's session times out, the next invocation raises `ConnectionLostException`. The client can recover from this situation by re-creating the session, re-creating the callback adapter, and adding all the callback servants to the Active Servant Map (ASM) of the re-created adapter.

See Also

- Bidirectional Connections
- Object Adapter Endpoints
- Object Identity
- Getting Started with Glacier2
- The Active Servant Map
- Glacier2.*

Glacier2 Application Class

The `Ice::Application` helper class encapsulates some basic Ice functionality such as communicator initialization, communicator destruction, and proper handling of signals and exceptions. The `Glacier2::Application` extends `Ice::Application` to add functionality that is commonly needed by Glacier2 clients:

- Keeps a session alive by periodically sending "heartbeat" requests
- Automatically restarts a session if a failure occurs
- Optionally creates an object adapter for callbacks
- Destroys the session when the application completes

Glacier2 Application is defined as follows :

C++11 C++98 C# Java Java Compat Python

```
namespace Glacier2
{
    class Application : public Ice::Application
    {
    public:

        using Ice::Application::Application;

        virtual std::shared_ptr<Glacier2::SessionPrx> createSession() =
0;
        virtual int runWithSession(int argc, char* argv[]) = 0;
        virtual void sessionDestroyed();

        static void restart();

        static std::shared_ptr<Glacier2::RouterPrx> router();
        static std::shared_ptr<Glacier2::SessionPrx> session();

        static std::string categoryForClient();
        static Ice::Identity createCallbackIdentity(const std::string&);
        static std::shared_ptr<Ice::ObjectPrx> addWithUUID(const
std::shared_ptr<Ice::Object>& servant);
        static std::shared_ptr<Ice::ObjectAdapter> objectAdapter();

        ...
    };
}
```

```
namespace Glacier2
{
    class Application : public Ice::Application
    {
    public:

        Application(Ice::SignalPolicy signalPolicy =
Ice::HandleSignals);

        virtual Glacier2::SessionPrx createSession() = 0;
        virtual int runWithSession(int argc, char* argv[]) = 0;
        virtual void sessionDestroyed();

        static void restart();

        static Glacier2::RouterPrx router();
        static Glacier2::SessionPrx session();

        static std::string categoryForClient();
        static Ice::Identity createCallbackIdentity(const std::string&);
        static Ice::ObjectPrx addWithUUID(const Ice::ObjectPtr&
servant);
        static Ice::ObjectAdapterPtr objectAdapter();

        ...
    };
}
```

```
namespace Glacier2
{
    public abstract class Application : Ice.Application
    {
        public Application(Ice.SignalPolicy signalPolicy =
Ice.SignalPolicy.HandleSignals) { ... }

        public abstract Glacier2.SessionPrx createSession();
        public abstract int runWithSession(string[] args);
        public virtual void sessionDestroyed() {}

        public static void restart() { ... }

        public static Glacier2.RouterPrx router() { ... }
        public static SessionPrx session() { ... }

        public static string categoryForClient() { ... }
        public static Ice.Identity createCallbackIdentity(string name) {
... }
        public static Ice.ObjectPrx addWithUUID(Ice.Object servant) {
... }
        public static Ice.ObjectAdapter objectAdapter() { ... }

        ...
    }
}
```

```
package com.zeroc.Glacier2;

public abstract class Application extends com.zeroc.Ice.Application
{
    public static class RestartSessionException extends Exception
    {
    }
    public Application() { ... }
    public Application(com.zeroc.Ice.SignalPolicy signalPolicy) { ... }

    public abstract com.zeroc.Glacier2.SessionPrx createSession();
    public abstract int runWithSession(String[] args) throws
RestartSessionException;
    public void sessionDestroyed() {}

    public static void restart() throws RestartSessionException { ... }

    public static com.zeroc.Glacier2.RouterPrx router() { ... }
    public static com.zeroc.Glacier2.SessionPrx session() { ... }

    public static String categoryForClient() throws
SessionNotExistException { ... }
    public static com.zeroc.Ice.Identity createCallbackIdentity(String
name) throws SessionNotExistException  { ... }
    public static com.zeroc.Ice.ObjectPrx
addWithUUID(com.zeroc.Ice.Object servant) throws
SessionNotExistException  { ... }
    public static com.zeroc.Ice.ObjectAdapter objectAdapter() throws
SessionNotExistException  { ... }

    ...
}
```

```
package Glacier2;

public abstract class Application extends Ice.Application
{
    public static class RestartSessionException extends Exception
    {
    }
    public Application() { ... }
    public Application(Ice.SignalPolicy signalPolicy) { ... }

    public abstract Glacier2.SessionPrx createSession();
    public abstract int runWithSession(String[] args) throws
RestartSessionException;
    public void sessionDestroyed() {}

    public static void restart() throws RestartSessionException { ... }

    public static Glacier2.RouterPrx router() { ... }
    public static Glacier2.SessionPrx session() { ... }

    public static String categoryForClient() throws
SessionNotExistException { ... }
    public static Ice.Identity createCallbackIdentity(String name)
throws SessionNotExistException  { ... }
    public static Ice.ObjectPrx addWithUUID(Ice.Object servant) throws
SessionNotExistException  { ... }
    public static Ice.ObjectAdapter objectAdapter() throws
SessionNotExistException  { ... }

    ...
}
```

```
# in Glacier2 module
class Application(Ice.Application):

    def __init__(self, signalPolicy=0):

    def createSession(self, args):
    def runWithSession(self, args):
    def sessionDestroyed(self):

    def restart(self):
    def router(self):
    def session(self):

    def categoryForClient(self):
    def createCallbackIdentity(self, name):
    def addWithUUID(self, servant):
    def objectAdapter(self):
```

The intent of this class is that you specialize Glacier2 `Application` and implement the pure virtual functions or abstract methods `createSession` and `runWithSession` in your derived class.

This Application class provides the following functions or methods:

- Constructor
  Like the Ice Application constructor, you can construct a Glacier2 application that handles signals (the default) or does not.

- `createSession`
  This pure virtual function or abstract method must be overridden by a subclass to create the application's Glacier2 session. A successful call to `createSession` is followed by a call to `runWithSession`. The application terminates if `createSession` throws an `Ice LocalException`.

- `runWithSession`
  This pure virtual function or abstract method must be overridden by a subclass and represents the "main loop" of the application. It is called after the communicator has been initialized and the Glacier2 session has been established. The arguments passed to `runWithSession` contain the arguments passed to `main` on `Application` with all Ice-related options removed. The implementation of `runWithSession` must return zero to indicate success and non-zero to indicate failure; the value returned by `runWithSession` becomes the return value of `main`.

  `runWithSession` can call `restart` to restart the session. This destroys the current session, creates a new session (by calling `createSession`), and calls `runWithSession` again. The `Application` base class also restarts the session if `runWithSession` throws any of the following Ice exceptions:
  - `ConnectionLostException`
  - `ConnectionRefusedException`
  - `RequestFailedException`
  - `TimeoutException`
  - `UnknownLocalException`

  All other exceptions cause the current session to be destroyed without restarting.

- `sessionDestroyed`
  A subclass can optionally override this function or method to take action when connectivity with the Glacier2 router is lost.

- `router`
  Returns the proxy for the Glacier2 router.

- `session`
  Returns the proxy for the current session.

- `restart`

Causes `Application` to destroy the current session, create a new session (by calling `createSession`), and start a new main loop (in `runWithSession`). This function or method does not return but rather throws a `RestartSessionException`, later caught by `Application`.

- `categoryForClient`
  Returns the category to be used in the identities of all of the client's callback objects. Clients must use this category for the router to forward callback requests to the intended client. This function or method raises `SessionNotExistException` if no session is currently active.
- `createCallbackIdentity`
  Creates a new Ice identity for a callback object with the given identity name.
- `addWithUUID`
  Adds a servant to the callback object adapter's Active Servant Map using a UUID for the identity name.
- `objectAdapter`
  Returns the object adapter used for callbacks, creating the object adapter if necessary.

Application and Threads
Just like Ice `Application`, Glacier2 `Application` should be considered single-threaded until you call `main`. Then, in `createSession` and `runWithSession`, you can call concurrently `Application`'s functions or methods from multiple threads. If you create threads, you must join them before or when `runWithSession` returns.

The thread you use to call `main` is used to initialize the `Communicator` and then to call `createSession` and later `runWithSession`. On Linux and macOS with C++, this thread must be the only thread of the process at the time you call `main` for signal handling to operate correctly.

See Also

- Application Helper Class
- Callbacks through Glacier2
- Glacier2 Session Management

Glacier2 SessionHelper Class

The "main loop" design imposed by the `Glacier2 Application` class is usually not suitable for graphical applications. Ice provides additional helper classes that better accommodate the needs of such applications:

- `SessionHelper`
  This class encapsulates a Glacier2 session and provides much of the same functionality as `Glacier2 Application`. The application must supply an instance of `SessionCallback` when creating the session; `SessionHelper` invokes this callback object when important events occur in the session's lifecycle.
- `SessionFactoryHelper`
  This class simplifies the task of creating a Glacier2 session. It provides overloaded `connect` methods that support the two authentication styles (user name/password and SSL credentials) accepted by the Glacier2 router, and returns an instance of `SessionHelper` for each new session.

- `SessionCallback`
  You implement this interface to receive notifications about session lifecycle events.

These classes are provided in the `Glacier2` namespace or package. We discuss them further in the subsections below.

Example
You can find sample applications that make use of these classes in the `Glacier2/simpleChat` directories of ice-demos.

The `SessionHelper` Class

The `SessionHelper` class encapsulates a Glacier2 session and keeps the session alive by periodically sending a hearbeat to the Glacier2 router. `SessionHelper` also provides several convenience functions or methods for common session-related actions:

C++11 C++98 C# Java Java Compat

```
namespace Glacier2
{
    class SessionHelper
    {
    public:
        virtual ~SessionHelper();

        virtual void destroy() = 0;
        virtual std::shared_ptr<Ice::Communicator> communicator() const
= 0;
        virtual std::string categoryForClient() const = 0;
        virtual std::shared_ptr<Ice::ObjectPrx> addWithUUID(const
std::shared_ptr<Ice::Object>&) = 0;
        virtual std::shared_ptr<SessionPrx> session() const = 0;
        virtual bool isConnected() const = 0;
        virtual std::shared_ptr<Ice::ObjectAdapter> objectAdapter() = 0;
    };
}
```

```
namespace Glacier2
{
    class SessionHelper : public virtual IceUtil::Shared
    {
    public:
        virtual ~SessionHelper();

        virtual void destroy() = 0;
        virtual Ice::CommunicatorPtr communicator() const = 0;
        virtual std::string categoryForClient() const = 0;
        virtual Ice::ObjectPrx addWithUUID(const Ice::ObjectPtr&) = 0;
        virtual SessionPrx session() const = 0;
        virtual bool isConnected() const = 0;
        virtual Ice::ObjectAdapterPtr objectAdapter() = 0;

        bool operator==(const SessionHelper&) const;
        bool operator!=(const SessionHelper&) const;
    };
    typedef IceUtil::Handle<SessionHelper> SessionHelperPtr;
}
```

```
namespace Glacier2
{
    public class SessionHelper
    {
        public void destroy() { ... }

        public Ice.Communicator communicator() { ... }

        public string categoryForClient() { ... }
        public Ice.ObjectPrx addWithUUID(Ice.Object servant) { ... }
        public SessionPrx session() { ... }
        public bool isConnected() { ... }
        public Ice.ObjectAdapter objectAdapter() { ... }

        ...
    }
}
```

```
package com.zeroc.Glacier2;

public class SessionHelper
{
    public void destroy() { ... }

    public com.zeroc.Ice.Communicator communicator() { ... }

    public String categoryForClient() throws SessionNotExistException {
... }
    public com.zeroc.Ice.ObjectPrx addWithUUID(com.zeroc.Ice.Object
servant) throws SessionNotExistException { ... }
    public SessionPrx session() { ... }
    public boolean isConnected() { ... }
    public com.zeroc.Ice.ObjectAdapter objectAdapter() throws
SessionNotExistException { ... }

    ...
}
```

```
package Glacier2;

public class SessionHelper
{
    public void destroy() { ... }

    public Ice.Communicator communicator() { ... }

    public String categoryForClient() throws SessionNotExistException {
... }
    public Ice.ObjectPrx addWithUUID(Ice.Object servant) throws
SessionNotExistException { ... }
    public SessionPrx session() { ... }
    public boolean isConnected() { ... }
    public Ice.ObjectAdapter objectAdapter() throws
SessionNotExistException { ... }

    ...
}
```

This class provides the following functions or methods:

- `destroy`
  Initiates the destruction of the Glacier2 session, including the communicator created by the `SessionHelper`. This is a non-blocking method that starts a background thread to perform potentially blocking activities. `SessionHelper` invokes the `disconnected` met hod on the application's callback object to indicate the completion of the `destroy` request. If the session has already been destroyed when `destroy` is called, `destroy` does nothing.
- `communicator`

Returns the communicator created by the `SessionHelper`.

- `categoryForClient`
  Returns the category that must be used in the identities of all callback objects. Raises `SessionNotExistException` if no session is currently active.
- `addWithUUID`
  Adds a servant to the callback object adapter using a UUID for the identity name. Raises `SessionNotExistException` if no session is currently active.
- `session`
  Returns a proxy for the Glacier2 session. Returns a nil proxy if no session is currently active.
- `isConnected`
  Returns true if the session is currently active, false otherwise.
- `objectAdapter`
  Returns the callback object adapter. This object adapter is only created if the session factory was configured to do so using `setUseCallbacks`. Raises `SessionNotExistException` if no session is currently active.

> The Java implementation of `SessionHelper` installs a JVM shutdown hook that calls `destroy` when the application is about to exit.

The `SessionFactoryHelper` Class

The `SessionFactoryHelper` class provides convenience functions or methods for configuring the settings that are commonly used to create a Glacier2 session, such as the router's host and port number. Once the application has completed its configuration, it calls one of the overloaded `connect` functions or methods to initialize a communicator, create a Glacier2 session, and receive a `SessionHelper` object with which it can manage the new session. An application should create a new `SessionFactoryHelper` object for each router instance that it uses.

`SessionFactoryHelper` creates an `InitializationData` object if the application does not pass one to the `SessionFactoryHelper` constructor. `SessionFactoryHelper` also creates a new property set if necessary, and then sets some configuration properties required by Glacier2 clients. The resulting `InitializationData` object is used in `connect` to create the new communicator.

C++11C++98C#JavaJava Compat

```
namespace Glacier2
{
    class SessionFactoryHelper : public
std::enable_shared_from_this<SessionFactoryHelper>
    {
    public:

        SessionFactoryHelper(const std::shared_ptr<SessionCallback>&
callback);
        SessionFactoryHelper(const Ice::InitializationData&, const
std::shared_ptr<SessionCallback>&);
        SessionFactoryHelper(const std::shared_ptr<Ice::Properties>&,
const std::shared_ptr<SessionCallback>&);

        ~SessionFactoryHelper();

        void destroy();

        void setRouterIdentity(const Ice::Identity&);
        Ice::Identity getRouterIdentity() const;

        void setRouterHost(const std::string&);
        std::string getRouterHost() const;

        void setProtocol(const std::string&);
        std::string getProtocol() const;

        void setTimeout(int);
        int getTimeout() const;

        void setPort(int port);
        int getPort() const;

        Ice::InitializationData getInitializationData() const;

        void setConnectContext(const std::map<std::string, std::string>&
context);

        void setUseCallbacks(bool);
        bool getUseCallbacks() const;

        SessionHelperPtr connect();
        SessionHelperPtr connect(const std::string& username,  const
std::string& password);

        ...
    };
}
```

```
namespace Glacier2
{
    class SessionFactoryHelper : public virtual IceUtil::Shared
    {
    public:

        SessionFactoryHelper(const SessionCallbackPtr& callback);
        SessionFactoryHelper(const Ice::InitializationData&, const
SessionCallbackPtr&);
        SessionFactoryHelper(const Ice::PropertiesPtr&, const
SessionCallbackPtr&);

        ~SessionFactoryHelper();

        void destroy();

        void setRouterIdentity(const Ice::Identity&);
        Ice::Identity getRouterIdentity() const;

        void setRouterHost(const std::string&);
        std::string getRouterHost() const;

        void setProtocol(const std::string&);
        std::string getProtocol() const;

        void setTimeout(int);
        int getTimeout() const;

        void setPort(int port);
        int getPort() const;

        Ice::InitializationData getInitializationData() const;

        void setConnectContext(const std::map<std::string, std::string>&
context);

        void setUseCallbacks(bool);
        bool getUseCallbacks() const;

        SessionHelperPtr connect();
        SessionHelperPtr connect(const std::string& username,  const
std::string& password);

        ...
    };
}
```

```
namespace Glacier2
{
    public class SessionFactoryHelper
    {
        public SessionFactoryHelper(SessionCallback callback) { ... }
        public SessionFactoryHelper(Ice.InitializationData initData,
SessionCallback callback) { ... }
        public SessionFactoryHelper(Ice.Properties properties,
SessionCallback callback) { ... }

        public void setRouterIdentity(Ice.Identity identity) { ... }
        public Ice.Identity getRouterIdentity() { ... }

        public void setRouterHost(string hostname) { ... }
        public string getRouterHost() { ... }

        public void setProtocol(string protocol) { ... }
        public string getProtocol() { ... }

        public void setTimeout(int timeoutMillisecs) { ... }
        public int getTimeout() { ... }

        public void setPort(int port) { ... }
        public int getPort() { ... }

        public Ice.InitializationData getInitializationData() { ... }
        public void setConnectContext(Dictionary<string, string>
context) { ... }

        public void setUseCallbacks(bool useCallbacks) { ... }
        public bool getUseCallbacks() { ... }

        public SessionHelper connect() { ... }
        public SessionHelper connect( string username,  string password)
{ ... }

        ...
    }
}
```

```
package com.zeroc.Glacier2;

public class SessionFactoryHelper
{
    public SessionFactoryHelper(SessionCallback callback) throws
com.zeroc.Ice.InitializationException;
    public SessionFactoryHelper(com.zeroc.Ice.InitializationData
initData, SessionCallback callback) throws
com.zeroc.Ice.InitializationException;
    public SessionFactoryHelper(com.zeroc.Ice.Properties properties,
SessionCallback callback) throws com.zeroc.Ice.InitializationException;

    public void setRouterIdentity(com.zeroc.Ice.Identity identity);
    public com.zeroc.Ice.Identity getRouterIdentity();

    public void setRouterHost(String hostname);
    public String getRouterHost();

    public void setProtocol(String protocol);
    public String getProtocol();

    public void setTimeout(int timeoutMillisecs);
    public int getTimeout();

    public void setPort(int port);
    public int getPort();

    public com.zeroc.Ice.InitializationData getInitializationData();

    public void setConnectContext(java.util.Map<String, String> ctx);

    public void setUseCallbacks(boolean useCallbacks);
    public boolean getUseCallbacks();

    public SessionHelper connect();
    public SessionHelper connect(String username, String password);
}
```

```
package Glacier2;

public class SessionFactoryHelper
{
    public SessionFactoryHelper(SessionCallback callback) throws
Ice.InitializationException;
    public SessionFactoryHelper(Ice.InitializationData initData,
SessionCallback callback) throws Ice.InitializationException;
    public SessionFactoryHelper(Ice.Properties properties,
SessionCallback callback) throws Ice.InitializationException;

    public void setRouterIdentity(Ice.Identity identity);
    public Ice.Identity getRouterIdentity();

    public void setRouterHost(String hostname);
    public String getRouterHost();

    public void setProtocol(String protocol);
    public String getProtocol();

    public void setTimeout(int timeoutMillisecs);
    public int getTimeout();

    public void setPort(int port);
    public int getPort();

    public Ice.InitializationData getInitializationData();

    public void setConnectContext(java.util.Map<String, String> ctx);

    public void setUseCallbacks(boolean useCallbacks);
    public boolean getUseCallbacks();

    public SessionHelper connect();
    public SessionHelper connect(String username, String password);
}
```

This class provides the following functions or methods:

- `SessionFactoryHelper` constructor with `SessionCallback` parameter
  This constructor is useful when your application has no other configuration requirements. The constructor allocates an `InitializationData` object and a new property set. The callback argument must not be null.
- `SessionFactoryHelper` constructor with `InitializationData` and `SessionCallback` parameters
  Use this constructor when you want to provide your own instance of `InitializationData`. The `callback` argument must not be null.
- `SessionFactoryHelper` constructor with `Properties` and `SessionCallback` parameters
  This constructor is convenient when you want to supply an initial set of properties. The callback argument must not be null.
- `setRouterIdentity`
  `getRouterIdentity`

Sets or gets the object identity of the Glacier2 router. If you don't call `setRouterIdentity` to provide an identity, the factory discovers the router's proxy by contacting the `RouterFinder` object using the endpoint information you've configured for the factory (host, port and so on).

> The factory ignores this setting if you configure `Ice.Default.Router`.

- `setRouterHost`
  `getRouterHost`
  Sets or gets the host name of the Glacier2 router. The default router host is `localhost`.

  > The factory ignores this setting if you configure `Ice.Default.Router`.

- `setProtocol`
  `getProtocol`
  Sets or gets the Ice protocol used for communications with the Glacier2 router. `protocol` can be `tcp`, `ssl`, `ws` or `wss`; the default protocol is `ssl`.

  > The factory ignores this setting if you configure `Ice.Default.Router`.

- `setTimeout`
  `getTimeout`
  Sets or gets the timeout in milliseconds for the connection to the Glacier2 router. No timeout is used if the argument is less than or equal to zero. The default timeout is 10,000 ms.

  > The factory ignores this setting if you configure `Ice.Default.Router`.

- `setPort`
  `getPort`
  Sets or gets the port on which the Glacier2 router is listening. The default is 4063 when the protocol is tcp or ws, and 4064 when the protocol is ssl or wss.

  > The factory ignores this setting if you configure `Ice.Default.Router`.

- `getInitializationData`
  Returns a reference to the `InitializationData` object that will be used during communicator initialization. If necessary, an application can make modifications to this object prior to calling `connect`.

- `setConnectContext`
  Sets the request context to be used when creating a session. This function or method must be invoked prior to `connect`.

- `setUseCallbacks`
  `getUseCallbacks`
  Determines whether the session helper automatically creates an object adapter for callback servants. The default behavior is to create the object adapter.

- `connect` with no parameters
  Initializes a communicator, creates a Glacier2 session using SSL credentials, and returns a new `SessionHelper` object. The `connected` function or method is called on the session callback if the session was created successfully, otherwise the callback's `connectFailed` function or method is called.

- `connect` with `username` and `password` parameters
  Initializes a communicator, creates a Glacier2 session using the given username and password, and returns a new `SessionHelper` object. The `connected` function or method is called on the session callback if the session was created successfully, otherwise the callback's `connectFailed` function or method is called.

The `SessionCallback` Interface

You need to supply an instance of `SessionCallback` when instantiating a `SessionFactoryHelper` object. The callback methods allow the application to receive notification about events in the lifecycle of the session:

C++11 C++98 C# Java Java Compat

```
namespace Glacier2
{
    class SessionCallback
    {
    public:
        virtual ~SessionCallback();
        virtual void createdCommunicator(const
std::shared_ptr<SessionHelper>& session) = 0;
        virtual void connected(const std::shared_ptr<SessionHelper>&) =
0;
        virtual void disconnected(const std::shared_ptr<SessionHelper>&)
= 0;
        virtual void connectFailed(const
std::shared_ptr<SessionHelper>&, const Ice::Exception&) = 0;
    };
}
```

```
namespace Glacier2
{
    class SessionCallback : public virtual IceUtil::Shared
    {
    public:
        virtual ~SessionCallback();
        virtual void createdCommunicator(const SessionHelperPtr&
session) = 0;
        virtual void connected(const SessionHelperPtr&) = 0;
        virtual void disconnected(const SessionHelperPtr&) = 0;
        virtual void connectFailed(const SessionHelperPtr&, const
Ice::Exception&) = 0;
    };
    typedef IceUtil::Handle<SessionCallback> SessionCallbackPtr;
}
```

```
namespace Glacier2
{
    public interface SessionCallback
    {
        void createdCommunicator(SessionHelper session);
        void connected(SessionHelper session);
        void disconnected(SessionHelper session);
        void connectFailed(SessionHelper session, Exception ex);
    }
}
```

```
package com.zeroc.Glacier2;

public interface SessionCallback
{
    void createdCommunicator(SessionHelper session);
    void connected(SessionHelper session) throws
SessionNotExistException;
    void disconnected(SessionHelper session);
    void connectFailed(SessionHelper session, Throwable ex);
}
```

```
package Glacier2;

public interface SessionCallback
{
    void createdCommunicator(SessionHelper session);
    void connected(SessionHelper session) throws
SessionNotExistException;
    void disconnected(SessionHelper session);
    void connectFailed(SessionHelper session, Throwable ex);
}
```

The following callback notifications are supported:

- `createdCommunicator`
  Called after successfully initializing a communicator.
- `connected`
  Called after successfully establishing the Glacier2 session. Your implementation of `connected` can raise `SessionNotExistException` to force the destruction of the new session.
- `disconnected`
  Called after the Glacier2 session is destroyed.
- `connectFailed`
  Called if a failure occurred while attempting to establish a Glacier2 session.

See Also

Securing a Glacier2 Router

As a firewall, a Glacier2 router represents a doorway into a private network, and in most cases that doorway should have a good lock. The obvious first step is to use SSL for the router's client endpoints. This allows you to secure the message traffic and restrict access to clients having the proper credentials. However, the router takes security even further by providing access control and filtering capabilities.

On this page:

Glacier2 Access Control

The authentication capabilities of SSL may not be sufficient for all applications: the certificate validation phase of the SSL handshake verifies that the user is who he says he is, but how do we know that he should be allowed to use the router? Glacier2 addresses this issue through the use of an access control facility that supports two forms of authentication: passwords and certificates. You can configure the router to use whichever authentication method is most appropriate for your application, or you can configure both methods in the same router.

Password Authentication

The router verifies the user name and password arguments to its `createSession` operation before it forwards any requests on behalf of the client. Given that the password is sent "in the clear," it is important to protect these values by using an SSL connection with the router.

There are two ways for the router to verify a user name and password. By default, the router uses a file-based access control list, but you can override this behavior by installing a proxy for an application-defined verifier object. Configuration properties define the password file name or the verifier proxy; if you install a verifier proxy, the password file is ignored. Since we have already discussed the password file, we will focus on the custom verifier interface here.

An application that has special requirements can implement the interface `Glacier2::PermissionsVerifier` to gain programmatic control over access to a router. This can be especially useful in situations where a repository of account information already exists (such as an LDAP directory), in which case duplicating that information in another file would be tedious and error-prone.

The Slice definition for the interface contains just one operation:

Slice

```
module Glacier2
{
    interface PermissionsVerifier
    {
        idempotent bool checkPermissions(string userId, string password,
out string reason)
            throws PermissionDeniedException;
    }
}
```

The router invokes `checkPermissions` on the verifier object, passing it the user name and password arguments that were given to `creat`

eSession. The operation must return true if the arguments are valid, and false otherwise. If the operation returns false, a reason can be provided in the output parameter. Starting with Ice 3.5, checkPermissions can also throw Glacier2::PermissionDeniedException. Glacier2 forwards this exception as-is to the client, which means the verifier can raise a subclass of PermissionDeniedException in order to provide more information to the client.

To configure a router with a custom verifier, set the configuration property Glacier2.PermissionsVerifier with the proxy for the object.

In situations where authentication is not necessary, such as during development or when running in a trusted environment, you can use Glacier2's built-in "null" permissions verifier. This object accepts any combination of user name and password, and you can enable it with the following property definition:

```
Glacier2.PermissionsVerifier=Glacier2/NullPermissionsVerifier
```

Note that the category of the object's identity (Glacier2 in this example) must match the value of the property Glacier2.InstanceName.

Example
A sample implementation of the PermissionsVerifier interface is provided in the demo/Glacier2/callback directory.

Certificate Authentication

The createSessionFromSecureConnection operation does not require a user name or password because the client's SSL connection to the router already supplies the credentials necessary to sufficiently identify the client, in the form of X.509 certificates.

It is up to you to decide what constitutes sufficient identification. For example, a single certificate could be shared by all clients if there is no need to distinguish between them, or you could generate a unique certificate for each client or a group of clients. Glacier2 does not enforce any particular policy, but simply delegates the decision of whether to accept the client's credentials to an application-defined object that implements the Glacier2::SSLPermissionsVerifier interface:

Slice

```
module Glacier2
{
    interface SSLPermissionsVerifier
    {
        idempotent bool authorize(SSLInfo info, out string reason)
throws PermissionDeniedException;
    }
}
```

Router clients may only use createSessionFromSecureConnection if the router is configured with a proxy for an SSLPermissionsVerifier object. The implementation of authorize must return true to allow the client to establish a session. To reject the session, authorize must return false and may optionally provide a value for reason, which is returned to the client as a member of PermissionDeniedException. Starting with Ice 3.5, it can also throw Glacier2::PermissionDeniedException. Glacier2 forwards this exception as-is to the client, which means the verifier can raise a subclass of PermissionDeniedException in order to provide more information to the client.

The verifier examines the members of SSLInfo to authenticate a client:

Slice

```
module Glacier2
{
    struct SSLInfo
    {
        string remoteHost;
        int remotePort;
        string localHost;
        int localPort;
        string cipher;
        Ice::StringSeq certs;
    }
}
```

The structure includes address information about the remote and local hosts, and a string that describes the ciphersuite negotiated for the SSL connection between the client and the router. These values are generally of interest for logging purposes, whereas the `certs` member supplies the information the verifier needs to make its decision. The client's certificate chain is represented as a sequence of strings that use the Privacy Enhanced Mail (PEM) encoding.

The first element of the sequence corresponds to the client's certificate, followed by its signing certificates. The certificate of the root Certificate Authority (CA) is the last element of the sequence. An empty sequence indicates that the client did not supply a certificate chain.

Although the certificate chain has already been validated by the SSL implementation, a verifier implementation typically needs to examine it in detail before making its decision. As a result, the verifier will need to convert the contents of `certs` into a more usable form. Some Ice platforms, such as Java and .NET, already provide certificate abstractions, and IceSSL supplies its own for C++ users. IceSSL for Java and .NET defines the method `IceSSL.Util.createCertificate`, which accepts a PEM-encoded string and returns an instance of the platform's certificate class. In C++, the class `IceSSL::Certificate` has a constructor that accepts a PEM-encoded string.

In addition to examining certificate attributes such as the distinguished name of the subject and issuer, it is also important that a verifier consider the length of the certificate chain.

To install your verifier, set the `Glacier2.SSLPermissionsVerifier` property with the proxy of your verifier object.

In situations where authentication is not necessary, such as during development or when running in a trusted environment, you can use Glacier2's built-in "null" permissions verifier. This object accepts the credentials of any client, and you can enable it with the following property definition:

```
Glacier2.SSLPermissionsVerifier=Glacier2/NullSSLPermissionsVerifier
```

Note that the category of the object's identity (`Glacier2` in this example) must match the value of the property `Glacier2.InstanceName`.

Interaction with a Permissions Verifier

The router attempts to contact the configured permissions verifiers at startup. If an object is unreachable, the router logs a warning message but continues its normal operation (you can suppress the warning using the `--nowarn` option.) The router does not contact a verifier again until it needs to invoke an operation on the object. For example, when a client asks the router to create a new session, the router makes another attempt to contact the verifier; if the object is still unavailable, the router logs a message and returns `PermissionDeniedException` to the client.

Obtaining SSL Credentials for a Router Client

Servers that wish to receive information about a client's SSL connection to the router can define the `Glacier2.AddConnectionContext` property. When enabled, the router adds several entries to the request context of each invocation it forwards to a server, providing

information such as the client's encoded certificate (if supplied) and addressing details. If the client's connection uses SSL, the router defines the `_con.peerCert` entry in the context. A server can check for the presence of this entry and also extract additional context entries as shown below in this example:

C++11 C++98

```
void unlockDoor(string id, const Ice::Current& current)
{
    auto i = current.ctx.find("_con.peerCert");
    if(i != current.ctx.end())
    {
        string certPEM;
        certPEM = i->second;
        cout << "Client address = "
             << current.ctx["_con.remoteAddress"]
             << ":" << current.ctx["_con.remotePort"] << endl;
        ...
    }
    ...
}
```

```
void unlockDoor(const string& id, const Ice::Current& current)
{
    Ice::Context::const_iterator i = current.ctx.find("_con.peerCert");
    if(i != current.ctx.end())
    {
        string certPEM;
        certPEM = i->second;
        cout << "Client address = "
             << current.ctx["_con.remoteAddress"]
             << ":" << current.ctx["_con.remotePort"] << endl;
        ...
    }
    ...
}
```

If the client supplied a certificate, the server can decode and examine it using the techniques discussed for IceSSL.

Request Filtering

The Glacier2 router is capable of filtering requests based on a variety of criteria, which helps to ensure that clients do not gain access to unintended objects.

Address Filters

To prevent a client from accessing arbitrary back-end hosts or ports, you can configure a Glacier2 router to validate the address information in each proxy that the client attempts to use. Two properties determine the router's filtering behavior:

- `Glacier2.Filter.Address.Accept`
  An address is accepted if it matches an entry in this property and does not match an entry in `Glacier2.Filter.Address.Reject`.

- `Glacier2.Filter.Address.Reject`
  An address is rejected if it matches an entry in this property.

The value of each property is a list of *address*:*port* pairs separated by spaces, as shown in the example below:

```
Glacier2.Filter.Address.Accept=192.168.1.5:4063 192.168.1.6:4063
```

This configuration allows clients to use only two hosts in the back-end network, and only one port on each host. A client that attempts to use a proxy containing any other host or port receives an `ObjectNotExistException` on its initial request.

You can also use ranges, groups and wildcards when defining your address filters. For example, the following property value shows how to use an address range:

```
Glacier2.Filter.Address.Accept=192.168.1.[5-6]:4063
```

This property is equivalent to the first example, but the range notation allows us to define the filter more concisely. Similarly, we can restate the property using the group notation by separating values with a comma:

```
Glacier2.Filter.Address.Accept=192.168.1.[5,6]:4063
```

The wildcard notation uses the * character to substitute for a value:

```
Glacier2.Filter.Address.Accept=10.0.*.1:4063
```

The range, group, and wildcard notation is also supported when specifying ports, as shown below:

```
Glacier2.Filter.Address.Accept=192.168.1.[5,6]:[10000-11000]
Glacier2.Filter.Address.Reject=192.168.1.[5,6]:[10500,10501]
```

In this configuration, the router allows clients to access all of the ports in the range `10000` to `11000`, except for the two ports `10500` and `10501`.

At first glance, you might think that the following property definition is pointless because it would prevent clients from accessing any back-end server:

```
Glacier2.Filter.Address.Reject=*
```

In reality, this configuration only prevents clients from accessing servers using direct proxies, that is, proxies that contain endpoints. As a result, the property causes Glacier2 to accept only indirect proxies.

> By default, a Glacier2 router forwards requests for any address, which is equivalent to defining the property `Glacier2.Filter.Address.Accept=*`.

Category Filters

The `Ice::Identity` type contains two string members: category and name. You can configure a router with a list of valid identity categories, in which case it only routes requests for objects in those categories. The configuration property `Glacier2.Filter.Category.Accept` supplies the category list:

```
Glacier2.Filter.Category.Accept=cat1 cat2
```

This property does not affect the routing of callback requests from back-end servers to router clients.

> By default a Glacier2 router forwards requests for any category.

If a category contains spaces, you can enclose the value in single or double quotes. If a category contains a quote character, it must be escaped with a leading backslash.

Glacier2 can optionally manipulate the category filter automatically. When you set `Glacier2.Filter.Category.AcceptUser` to a value of 1, the router adds the session's user name (for password authentication) or distinguished name (for SSL authentication) to the list of accepted categories. To ensure the uniqueness of your categories, you may prefer setting the property to a value of 2, which causes the router to prepend an underscore to the user name or distinguished name before adding it to the list.

A session manager can also configure category filters dynamically using Glacier2's `SessionControl` interface.

Identity Filters

The ability to filter on identity categories, as described in the previous section, is a convenient way to limit clients to particular groups of objects. For even stricter control over the identities that clients are allowed to access, you can use the `Glacier2.Filter.Identity.Accept` property. The value of this property is a list of identities, separated by whitespace, representing the *only* objects the router's clients may use.

If an identity contains spaces, you can enclose the value in single or double quotes. If an identity contains a quote character, it must be escaped with a leading backslash.

Clearly, specifying a static list of identities is only practical for a small set of objects. Furthermore, in many applications, the complete set of identities cannot be known in advance, such as when objects are created on a per-session basis and use UUIDs in their identities. For these situations, category-based filtering is generally sufficient. However, a session manager can also use Glacier2's dynamic filtering interface, `SessionControl`, to manage the set of valid identities at run time.

Adapter Filters

Applications often use IceGrid in their back-end network to simplify server administration and take advantage of the benefits offered by indirect proxies. Once you have configured Glacier2 with an appropriate locator proxy, clients can use indirect proxies to refer to objects in IceGrid-managed servers. Indirect proxies come in two forms: one that contains only an identity, and one that contains an identity and an object adapter identifier. You can use the category and identity filters described in previous sections to control identity-only proxies, and you can use the property `Glacier2.Filter.AdapterId.Accept` to enforce restrictions on indirect proxies that use an object adapter identifier.

For example, the following property definition allows a client to use the proxy `factory@WidgetAdapter` but not the proxy `factory@SecretAdapter`:

```
Glacier2.Filter.AdapterId.Accept=WidgetAdapter
```

If an adapter identifier contains spaces, you can enclose the value in single or double quotes. If an adapter identifier contains a quote character, it must be escaped with a leading backslash.

A session manager can also configure this filter dynamically using Glacier2's `SessionControl` interface.

Proxy Filters

The Glacier2 router maintains an internal routing table that contains an entry for each proxy used by a router client; the size of the routing table grows in proportion to the number of clients and their proxy usage. Furthermore, the amount of memory that the routing table consumes is affected by the number of endpoints in each proxy. Glacier2 provides two properties that you can use to limit the size of the routing table and defend against malicious router clients.

The property `Glacier2.RoutingTable.MaxSize` specifies the maximum number of entries allowed in the routing table. If the size of the table exceeds the value of this property, the router evicts older entries on a least-recently-used basis. (Eviction of proxies from the routing table is transparent to router clients.) The default size of the routing table is 1000, but you may need to define a different value depending on the needs of your application. While experimenting with different values, you may find it useful to define the property `Glacier2.Trace.RoutingTable` to see a log of the router's activities with respect to the routing table.

The property `Glacier2.Filter.ProxySizeMax` sets a limit on the size of a stringified proxy. The Ice run time places no limits on the size of proxy components such as identities and host names, but a malicious client could manufacture very large proxies in a denial-of-service attack on a Glacier2 router. By setting this property to a reasonably small value, you can prevent proxies from consuming excessive memory in the router process.

Client Impact

The Glacier2 router immediately terminates a client's session if it attempts to use a proxy that is rejected by an address filter or exceeds the size limit defined by the property `Glacier2.Filter.ProxySizeMax`. The Ice run time in the client responds by raising `ConnectionLostException` to the application.

For category, identity, and adapter identifier filters, the router raises `ObjectNotExistException` if any of the filters rejects a proxy and none of the filters accepts it.

To obtain more information on the router's reasons for terminating a session or rejecting a request, set the following property and examine the router's log output:

```
Glacier2.Client.Trace.Reject=1
```

Glacier2 Routing Table

The Glacier2 router maintains an internal routing table for each session. The routing table holds every proxy used by its session. Consequently, the size of the routing table grows in proportion to the number of proxies used by a session. Furthermore, the amount of memory that all of the routing tables consume grows with the number of active sessions.

The property `Glacier2.RoutingTable.MaxSize` allows you to specify an upper limit on the number of entries in the routing table. If the size of the table exceeds the value of this property, the router evicts older entries on a least-recently-used basis. (Eviction of proxies from the routing table is transparent to router clients.) The default size of the routing table is 1000, but you may need to define a different value depending on the needs of your application. While experimenting with different values, you may find it useful to define the property `Glacier2.Trace.RoutingTable` to see a log of the router's activities with respect to the routing table.

The router does not remove entries from a session's routing table except when evicting an old entry to make room for a new one. In particular, an exception that occurs while routing a request for a proxy does *not* cause that proxy to be removed from the routing table. Note however that the routing table is destroyed upon session destruction.

Glacier2 Administrative Interface

Glacier2 supports an administrative interface that allows you to shut down a router programmatically:

Slice

```
module Glacier2
{
    interface Admin
    {
        idempotent void shutdown();
    }
}
```

To prevent unwanted clients from using the `Admin` interface, the object is only accessible on the endpoints defined by the `Glacier2.Admi`

`n.Endpoints` property. This property has no default value, meaning you must define the property in order to make the `Admin` object accessible.

If you decide to define `Glacier2.Admin.Endpoints`, choose your endpoints carefully. We generally recommend the use of endpoints that are accessible only from behind a firewall.

See Also

- Glacier2.*
- IceSSL
- Getting Started with Glacier2
- Callbacks through Glacier2
- Dynamic Request Filtering with Glacier2
- IceGrid and Glacier2 Integration

Glacier2 Session Management

A Glacier2 router requires a client to create a session and forwards requests on behalf of the client until its session expires. A session expires when it is explicitly destroyed, or when it times out due to inactivity.

You can configure a router to use a custom session manager if your application needs to track the router's session activities. For example, your application may need to acquire resources and initialize the state of back-end services for each new session, and later reclaim those resources when the session expires.

As with the authentication facility, Glacier2 provides two session manager interfaces that an application can implement. The `SessionManager` interface receives notifications about sessions that use password authentication, while the `SSLSessionManager` interface is for sessions authenticated using SSL certificates.

On this page:

Glacier2 Session Manager Interfaces

The relevant Slice definitions are shown below:

Slice

```
module Glacier2
{
    exception CannotCreateSessionException
    {
        string reason;
    }

    interface Session
    {
        void destroy();
    }

    interface SessionManager
    {
        Session* create(string userId, SessionControl* control)
            throws CannotCreateSessionException;
    }

    interface SSLSessionManager
    {
        Session* create(SSLInfo info, SessionControl* control)
            throws CannotCreateSessionException;
    }
}
```

When a client creates a session by invoking `createSession` on the `Router` interface, the router validates the client's user name and password and then calls `SessionManager::create`. Similarly, a call to `createSessionFromSecureConnection` causes the router to invoke `SSLSessionManager::create`. The `SSLInfo` structure provides details about the client's SSL connection. The second argument to the `create` operations is a proxy for a `SessionControl` object, which a session can use to perform dynamic filtering.

The `create` operations must return the proxy of a new `Session` object, or raise `CannotCreateSessionException` and provide an appropriate reason. The `Session` proxy returned by `create` is ultimately returned to the client as the result of `createSession` or `createSessionFromSecureConnection`.

Glacier2 invokes the `destroy` operation on a `Session` proxy when the session expires, giving a custom session manager the opportunity to reclaim resources that were acquired for the session during `create`.

> The `create` operations may be called with information that identifies an existing session. For example, this can occur when a client loses its connection to the router but its previous session has not yet expired (and therefore the router has not yet invoked `destroy` on its `Session` proxy). A session manager implementation must be prepared to handle this situation.

To configure the router with a custom session manager, define the properties `Glacier2.SessionManager` or `Glacier2.SSLSessionManager` with the proxies of the session manager objects. If necessary, you can configure a router with proxies for both types of session managers. If a session manager proxy is not supplied, the call to `createSession` or `createSessionFromSecureConnection` always returns a null proxy.

The router attempts to contact the configured session manager at startup. If the object is unreachable, the router logs a warning message but continues its normal operation (you can suppress the warning using the `--nowarn` option). The router does not contact the session manager again until it needs to invoke an operation on the object. For example, when a client asks the router to create a new session, the router makes another attempt to contact the session manager; if the session manager is still unavailable, the router logs a message and returns `CannotCreateSessionException` to the client.

> **Example**
> A sample implementation of the `SessionManager` interface is provided in the `demo/Glacier2/callback` directory.

Glacier2 Session Timeouts

The value of the `Glacier2.SessionTimeout` property specifies the number of seconds a session must be inactive before it expires. This property is not defined by default, which means sessions never expire due to inactivity. If a non-zero value is specified, it's very important that the application chooses a value that does not result in premature session expiration. For example, if it's normal for a client to create a session and then have long periods of inactivity, then a suitably long timeout must be chosen, or the connection must have periodic activity, or timeouts must be disabled altogether. Callbacks from a server to the client also keep a session alive, in which case Glacier2 automatically destroys the session if a failure occurs while forwarding a server callback to the client.

Once a session has expired (or been destroyed for some other reason), the client will no longer be able to send requests via the router and receives a `ConnectionLostException` for each subsequent invocation. The client must explicitly create a new session in order to continue using the router.

In general, we recommend the use of an appropriate session timeout, otherwise resources created for each session will accumulate in the router.

> The easiest way for a client to keep a session alive is to enable ACM heartbeats.

Connection Timeouts on Routed Proxies

The timeout for a client's connection to a Glacier2 router works the same as any other connection timeout: in addition to the timeout specified in a proxy's endpoints, several properties can also influence the timeout. For the sake of this discussion, let's ignore the properties and focus on the proxy endpoints.

Before we discuss timeouts, we need to briefly review some Glacier2 concepts. When a Glacier2 client configures a router proxy, normally by setting `Ice.Default.Router`, it causes all proxies to become *routed proxies*. For an invocation on a routed proxy, the Ice run time in the client ignores that proxy's endpoints and instead forwards the invocation to the router for delivery.

There are two kinds of proxies here: the router proxy, and routed proxies. It's important to recognize this distinction in order to understand the following timeout semantics:

- The endpoint timeout in the **router proxy** determines the connection timeout for the connection between the client and the router.

Suppose we use the following settings:

```
Ice.Default.Router=Glacier2/router:tcp -h routerHost -p 4063 -t
10000
MyProxy=hello:tcp -h backEndHost -p 8888 -t 5000
```

This setting configures a ten second timeout for the client's connection to the router. Endpoint timeouts in routed proxies have no effect on this connection.

- The endpoint timeout in a **routed proxy** affects the router's connection to a back-end server. The Ice run time in the client internally registers routed proxies with the router because the router uses these proxies when it forwards invocations from the client to the desired back-end server. For the routed proxy `MyProxy` in the example above, the router would establish its own connection to port 8888 on `backEndHost` and configure a timeout of five seconds for that connection.

Again, it's critically important that the client's connection to the router remain open for as long as the client requires its session, as closing this connection effectively terminates the session. Since Ice interprets the occurrence of a connection timeout as a hard error and closes the offending connection, select your timeout for the router proxy with care.

> As of Ice 3.6, a Glacier2 client can safely use invocation timeouts and active connection management.

For invocations made by Glacier2 to a back-end server, whatever timeout value is set on the *first* proxy that is used to make an invocation applies to all proxies for the same object. This is necessary because Glacier2 adds the proxy to its routing table during the first invocation and, thereafter, reuses that cached proxy for all invocations to the same object identity. Here is an example to illustrate this:

C++11 C++98

```
// 10-second timeout for the connection to the router.
auto router = communicator->stringToProxy("Glacier2/router:tcp -h
routerHost -p 4063 -t 10000");
communicator->setDefaultRouter(Ice::uncheckedCast<RouterPrx>(router));

// Ping back-end server with 20-second timeout
communicator->stringToProxy("id:tcp -h backEndHost -p 8888 -t
20000")->ice_ping();

// Ping back-end server with 30-second timeout
communicator->stringToProxy("id:tcp -h backEndHost -p 8888 -t
30000")->ice_ping();
```

```
// 10-second timeout for the connection to the router.
ObjectPrx router = communicator->stringToProxy("Glacier2/router:tcp -h
routerHost -p 4063 -t 10000");
communicator->setDefaultRouter(RouterPrx::uncheckedCast(router));

// Ping back-end server with 20-second timeout
communicator->stringToProxy("id:tcp -h backEndHost -p 8888 -t
20000")->ice_ping();

// Ping back-end server with 30-second timeout
communicator->stringToProxy("id:tcp -h backEndHost -p 8888 -t
30000")->ice_ping();
```

The first call to `ice_ping`, when forwarded by Glacier2 to the server, establishes a connection with a 20-second timeout. The second call to `ice_ping` reuses the existing connection with a 20-second timeout, even though the proxy specifies a 30-second timeout.

Connection Caching for Session Managers

You can distribute the load among multiple session manager objects by configuring the router with a session manager proxy that contains multiple endpoints. Glacier2 disables connection caching on this proxy so that each invocation on a session manager attempts to use a different endpoint.

This behavior achieves a basic form of load balancing without depending on the replication features provided by IceGrid. Be aware that including an invalid endpoint in your session manager proxy, such as the endpoint of a session manager server that is not currently running, can cause router clients to experience delays during session creation.

If your session managers are in an IceGrid replica group, refer to IceGrid and Glacier2 Integration for more information on the router's caching behavior.

See Also

- Getting Started with Glacier2
- Securing a Glacier2 Router
- Dynamic Request Filtering with Glacier2
- IceGrid and Glacier2 Integration
- Object Adapter Replication
- Glacier2.*

Dynamic Request Filtering with Glacier2

Glacier2 can be statically configured to filter requests, and also allows a session manager to customize filters for each session at run time via its `SessionControl` interface:

Slice

```
module Glacier2
{
    interface SessionControl
    {
        StringSet* categories();
        StringSet* adapterIds();
        IdentitySet* identities();
        void destroy();
    }
}
```

The router creates a `SessionControl` object for each client session and supplies a proxy for the object to the session manager `create` operations. Note that the `SessionControl` proxy is null unless the router is configured with server endpoints.

Invoking the `destroy` operation causes the router to destroy the client's session, which eventually results in an invocation of `destroy` on the application-defined `Session` object, if one was provided.

The interface operations `categories`, `adapterIds`, and `identities` return proxies to objects representing the modifiable filters for the session. The router initializes these filters using their respective static configuration properties.

The `SessionControl` object uses a `StringSet` to manage the category and adapter identifier filters:

Slice

```
module Glacier2
{
    interface StringSet
    {
        idempotent void add(Ice::StringSeq additions);
        idempotent void remove(Ice::StringSeq deletions);
        idempotent Ice::StringSeq get();
    }
}
```

Similarly, the `IdentitySet` interface manages the identity filters:

Slice

```
module Glacier2
{
    interface IdentitySet
    {
        idempotent void add(Ice::IdentitySeq additions);
        idempotent void remove(Ice::IdentitySeq deletions);
        idempotent Ice::IdentitySeq get();
    }
}
```

In both interfaces, the `add` operation silently ignores duplicates, and the `remove` operation silently ignores non-existent entries.

Dynamic filtering is often necessary when each session must be restricted to a particular group of objects. Upon session creation, a session manager typically allocates a number of objects in back-end servers for that session to use. To prevent other sessions from accessing these objects (intentionally or not), the session manager can configure the session's filters so that it is only permitted to use the objects that were created for it.

For example, a session manager can retain the `SessionControl` proxy and add a new identity to the `IdentitySet` as each new object is created for the session. A simpler solution is to create a unique category for the session, add it to the session's category filter, and use that category in the identities of all objects accessible to that session. Using a category filter in this way reserves an identity namespace for each session and avoids the need to update the filter for each new object.

To aid in logging and debugging, you can select a category that identifies the client, such as the user name that was supplied during session creation, or an attribute of the client's certificate such as the common name, as long as the selected category is sufficiently unique that it will not conflict with another client's session. You must also ensure that the categories you assign to sessions never match the categories of back-end objects that are not meant to be accessed by router clients. As an example, consider the following session manager implementation:

C++11 C++98

```
class SessionManagerI : public Glacier2::SessionManager
{
public:

    virtual shared_ptr<Glacier2::SessionPrx>
    create(string username, shared_ptr<Glacier2::SessionControlPrx>
ctrl, const Ice::Current& current)
    {
        string category = "_" + username;
        ctrl->categories()->add(category);
        // ...
    }
};
```

```
class SessionManagerI : public Glacier2::SessionManager
{
public:

    virtual Glacier2::SessionPrx
    create(const string& username, const Glacier2::SessionControlPrx&
ctrl, const Ice::Current& current)
    {
        string category = "_" + username;
        ctrl->categories()->add(category);
        // ...
    }
};
```

This session manager derives a category for the session by prepending an underscore to the user name and then adds this category to the session's filter. As long as our back-end objects do not use a leading underscore in their identity categories, this strategy guarantees that a session's category can never match the category of a back-end object.

For your convenience, Glacier2 already includes support for automatic category filtering.

See Also

- Callbacks through Glacier2
- Securing a Glacier2 Router
- Glacier2 Session Management

Glacier2 Request Buffering

A Glacier2 router can forward requests in buffered or unbuffered mode. In addition, the buffering mode can be set independently for each direction (client-to-server and server-to-client).

The configuration properties `Glacier2.Client.Buffered` and `Glacier2.Server.Buffered` govern the buffering behavior. The former affects buffering of requests from clients to servers, and the latter affects buffering of requests from servers to clients. If a property is not specified, the default value is `1`, which enables buffering. A property value of `0` selects the unbuffered mode.

The primary difference between the two modes is in the way requests are forwarded:

- Buffered
  The router queues incoming requests and outgoing replies for delivery in a separate thread.
- Unbuffered
  The router forwards requests in the same thread that received the request.

Although unbuffered mode consumes fewer resources than buffered mode, certain features such as request overriding and request batching are available only in buffered mode.

See Also

- How Glacier2 uses Request Contexts

How Glacier2 uses Request Contexts

The Glacier2 router examines the context of an incoming request for special keys that affect how the router forwards the request. These contexts have the same semantics regardless of whether the request is sent from client to server or from server to client.

On this page:

The `_fwd` Context

The `_fwd` context determines the proxy mode that the router uses when forwarding the request. The value associated with the `_fwd` key must be a string containing one or more of the characters shown in the following table:

Value	Mode
d	datagram
D	Batch datagram
o	Oneway
O	Batch oneway
s	Secure
t	Twoway
z	Compress

Legal values for the `_fwd` context key.

These characters match the corresponding stringified proxy options.

For requests whose `_fwd` context specify a batch mode, the forwarding behavior of the router depends on whether it is batching requests.

If the `_fwd` key is not present in a request context, the mode used by the router to forward that request depends on the request ID:

- If the sender sent the request as twoway (request ID > 0), the router also uses twoway mode.
- If the sender sent the request as a oneway or batch oneway (request ID == 0), the router's behavior is determined by its configuration properties.

The `_ovrd` Context

In buffered mode, the router allows a new incoming request to override any pending requests that are still in the router's buffer, effectively replacing any pending requests with the new request.

For a new request to override a pending request, both requests must meet the following criteria:

- they specify the `_ovrd` key in the request context with the same value
- they are oneway requests
- they are requests on the same object.

This feature is intended to be used by senders that are sending frequent oneway requests in which the most recent request takes precedence. This feature minimizes the number of requests that are forwarded to a given object when requests are sent frequently enough that they accumulate in the router's buffer before the router has a chance to process them.

Note that the properties `Glacier2.Client.SleepTime` and `Glacier2.Server.SleepTime` can be used to add a delay to the router once it has sent all pending requests. Setting a delay increases the likelihood of overrides actually taking effect. These properties are described in the next section.

Glacier2 provides complete isolation between client sessions: requests coming from different client sessions never overwrite one another.

Forwarding Batch Requests

A sender can direct the router to forward oneway requests in batches by including the `D` or `O` characters in the `_fwd` context. If the router is configured for buffered mode and several such requests accumulate in its buffer, the router forwards them together in a batch rather than as individual requests.

In addition, the properties `Glacier2.Client.AlwaysBatch` and `Glacier2.Server.AlwaysBatch` determine whether oneway requests are always batched regardless of the `_fwd` context. The former property affects requests from clients to servers, while the latter affects requests from servers to clients. If a property is defined with a non-zero value, then all requests whose `_fwd` context includes the `o` character or were sent as oneway invocations are treated as if `O` were specified instead, and are batched when possible. Likewise, requests whose `_fwd` context includes the `d` character or were sent as datagram invocations are treated as if `D` were specified. If a property is not defined, the router only batches requests if specifically directed to do so by the `_fwd` context.

The configuration properties `Glacier2.Client.SleepTime` and `Glacier2.Server.SleepTime` can be used to force the router's delivery threads to sleep for the specified number of milliseconds after the router has sent all of its pending requests. (Incoming requests are buffered during this period.) The delay is useful to increase the effectiveness of batching because it makes it more likely for additional requests to accumulate in a batch before the batch is sent. If these properties are not defined, or their value is zero, the corresponding thread does not sleep after sending buffered requests.

Context Forwarding

The configuration properties `Glacier2.Client.ForwardContext` and `Glacier2.Server.ForwardContext` determine whether the router includes the context when forwarding a request. The former property affects requests from clients to servers, while the latter affects requests from servers to clients. If a property is not defined or has the value zero, the router does not include the context when forwarding requests.

The configuration property Glacier2.AddConnectionContext determines whether the router includes the connection information in the context when forwarding a request.

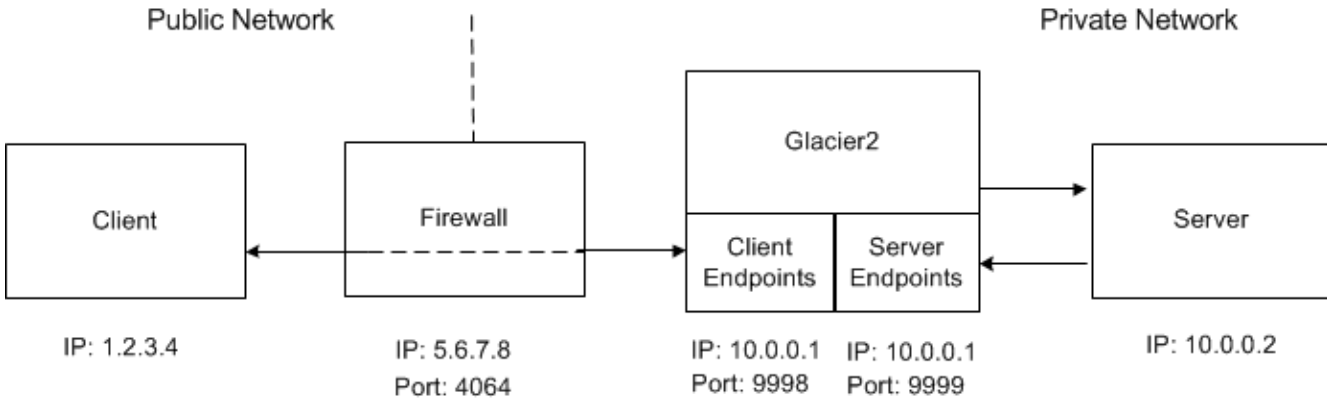
See Also

- Request Contexts
- Proxy and Endpoint Syntax
- Batched Invocations
- Glacier2.*

Configuring Glacier2 behind an External Firewall

The Glacier2 router requires only one external port to receive connections from clients and therefore can easily coexist with a network firewall device.

For example, consider the network shown in the following illustration:



The Glacier2 router in the example above has both of its endpoints in the private network and its host requires only one IP address, unlike the example we showed in the discussion of bidirectional connections in which the Glacier2 host straddled both networks.

We assume that the firewall has been configured to forward connections from port 4064 to the router's client endpoint at port 9998. Meanwhile, the client must be configured to use the firewall's address information in its router proxy, as shown below:

```
Ice.Default.Router=Glacier2/router:ssl -h 5.6.7.8 -p 4064
```

The Glacier2 router configuration for this example requires the following properties:

```
Glacier2.Client.Endpoints=ssl -h 10.0.0.1 -p 9998
Glacier2.Client.PublishedEndpoints=ssl -h 5.6.7.8 -p 4064
Glacier2.Server.Endpoints=tcp -h 10.0.0.1 -p 9999
```

We need to specify published endpoints for the client object adapter because the router is located behind a firewall. Without this property, any proxies that the router creates would use the endpoints specified in `Glacier2.Client.Endpoints`, but of course those endpoints are inaccessible to clients outside the firewall. The `PublishedEndpoints` property forces the Ice run time to use the given endpoints in proxies created by the client object adapter.

Note also that the server endpoint in this example includes a fixed port `(9999)`, but a fixed port is not required in the server endpoint for the router to operate properly.

See Also

- Callbacks through Glacier2
- Object Adapter Endpoints
- Glacier2.*

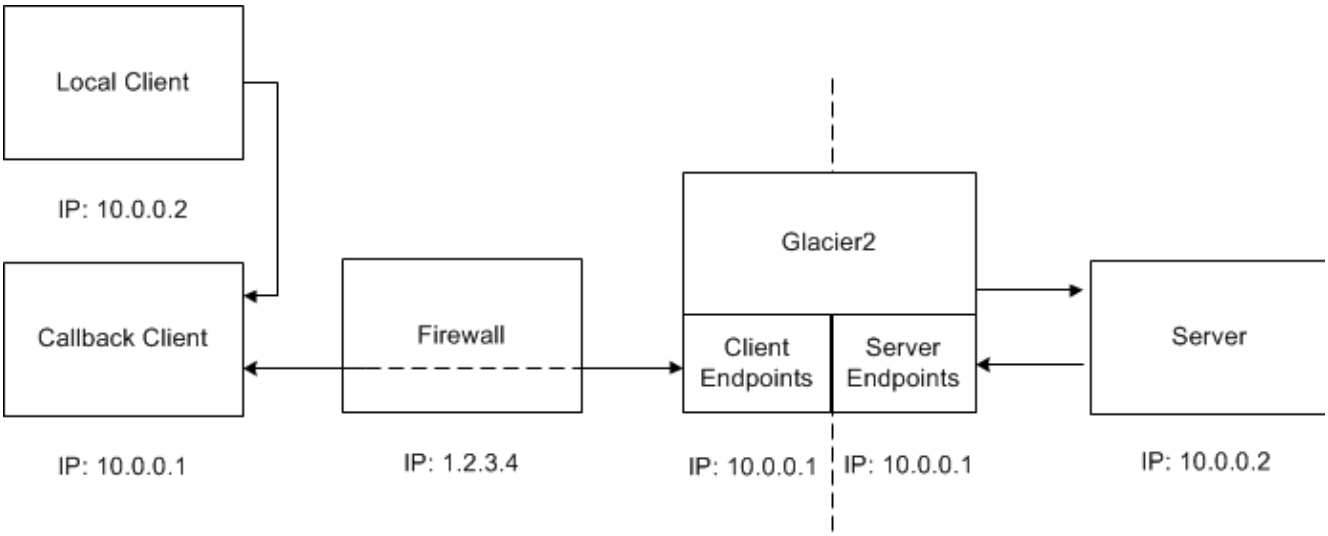
Advanced Glacier2 Client Configurations

This section details strategies that Glacier2 clients can use to address more advanced requirements.

On this page:

Callback Strategies with Multiple Object Adapters

An application that needs to support callback requests from a router as well as requests from local clients should use multiple object adapters to ensure that proxies created by these object adapters contain the appropriate endpoints. For example, suppose we have the network configuration as shown in the following illustration:



Notice that the two local area networks use the same private network addresses, which is not an unrealistic scenario.

Now, if the callback client were to use a single object adapter for handling both callback requests and local requests, then any proxies created by that object adapter would contain the application's local endpoints as well as the router's server endpoints. As you might imagine, this could cause some subtle problems.

1. When the local client attempts to establish a connection to the callback client via one of these proxies, it might arbitrarily select one of the router's server endpoints to try first. Since the router's server endpoints use addresses in the same network, the local client attempts to make a connection over the local network, with two possible outcomes: the connection attempts to those endpoints fail, in which case they are skipped and the real local endpoints are attempted; or, even worse, one of the endpoints might accidentally be valid in the local network, in which case the local client has just connected to some unintended server.
2. The server may encounter similar problems when attempting to establish a local connection to the router in order to make a callback request.

The solution is to dedicate an object adapter solely to handling callback requests, and another one for servicing local clients. The object adapter dedicated to callback requests must be configured with the router proxy.

Using Multiple Routers

A client is not limited to using only one router at a time: the proxy method `ice_router` allows a client to configure its routed proxies as necessary. With respect to callbacks, a client must create a new callback object adapter for each router that can forward callback requests to the client. A client must also be aware of the object identities in use by the routers.

Using the `RouterFinder` Interface

A router's identity can be changed by setting the `Glacier2.InstanceName` property, which affects the category portion of the identity. The default identity of a Glacier2 router is `Glacier2/Router`, but we can change it to `Production/Router` with the following setting:

```
Glacier2.InstanceName=Production
```

A client could configure its corresponding router proxy as follows:

```
Ice.Default.Router=Production/Glacier2:tcp -p 4063 -h prodhost
```

In most cases the client can statically configure the router's proxy as we've shown here. For clients that need to discover a router's proxy at run time, Ice also requires router implementations to support the `RouterFinder` interface:

Slice

```
module Ice
{
    interface RouterFinder
    {
        Router* getRouter();
    }
}
```

An object supporting this interface must be available with the identity `Ice/RouterFinder`. By knowing the host and port of a router's client endpoints, a client can discover the router's proxy with a call to `getRouter`:

C++11 C++98

```
auto prx = communicator->stringToProxy("Ice/RouterFinder:tcp -p 4063 -h
prodhost");
auto finder = Ice::checkedCast<Ice::RouterFinderPrx>(prx);
auto router = finder->getRouter();
communicator->setDefaultRouter(router);
```

```
Ice::ObjectPrx prx = communicator->stringToProxy("Ice/RouterFinder:tcp
-p 4063 -h prodhost");
Ice::RouterFinderPrx finder = Ice::RouterFinderPrx::checkedCast(prx);
Ice::RouterPrx router = finder->getRouter();
communicator->setDefaultRouter(router);
```

See Also

- Getting Started with Glacier2
- Callbacks through Glacier2
- Proxy Methods

IceGrid and Glacier2 Integration

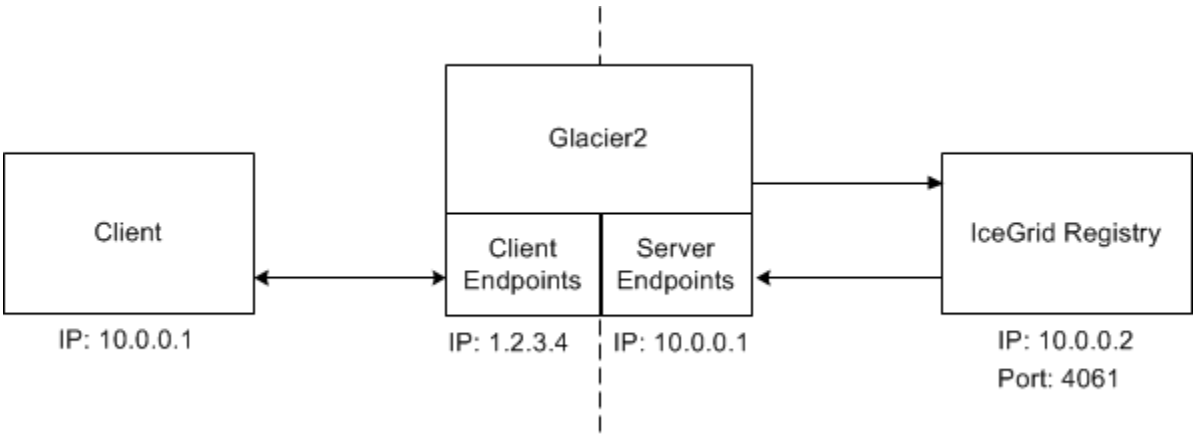
IceGrid is a server activation and location service. This section describes the ways in which you can integrate Glacier2 and IceGrid.

On this page:

Configuring Router Clients for IceGrid

It is not uncommon for a Glacier2 client to require access to a locator service such as IceGrid. In the absence of Glacier2, a locator client would typically define the property `Ice.Default.Locator` with a stringified proxy for the locator service. However, when that locator service is accessed via a Glacier2 router, the configuration requirements are slightly different. It is no longer necessary for the client's configuration to include `Ice.Default.Locator`; this property must be defined in the router's configuration instead.

For example, consider the following network architecture:



In this case the Glacier2 router's configuration must include the property shown below:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h 10.0.0.2 -p 4061
```

Using Replicated Session Managers

An IceGrid application might want to use replication to increase the availability of Glacier2 session managers. When you configure an indirect proxy for a session manager (and configure Glacier2 with a locator proxy), the Ice run time in the router queries the locator to obtain a proxy for a session manager replica.

By default, this proxy is cached for 10 minutes, meaning the router uses the same session manager proxy to create sessions for a 10-minute period, after which it queries the locator again. If you want to distribute the session-creation load among the session manager replicas more evenly, you can decrease the locator cache timeout using configuration properties.

For example, the following settings use a timeout of 30 seconds:

```
Glacier2.SessionManager.LocatorCacheTimeout=30
Glacier2.SSLSessionManager.LocatorCacheTimeout=30
```

As you can see, timeouts are specified individually for the `SessionManager` and `SSLSessionManager` proxies. You can also disable

caching completely by using a value of 0, in which case the router queries the locator before every invocation on a session manager. See the discussion of session management for more details.

See Also

- IceGrid
- Glacier2 Session Management
- Getting Started with IceGrid
- Object Adapter Replication

Glacier2 Metrics

You can monitor Glacier2 using the Administrative Facility and the Metrics Facet. Glacier2 provides a metrics class to monitor session related metrics. The Glacier2 session metrics class is defined in `Glacier2/Metrics.ice` and is shown below.

Slice

```
namespace IceMX
{
    class SessionMetrics extends Metrics
    {
        int forwardedClient = 0;
        int forwardedServer = 0;
        int routingTableSize = 0;
        int queuedClient = 0;
        int queuedServer = 0;
        int overriddenClient = 0;
        int overriddenServer = 0;
    }
}
```

Glacier2 records session metrics in the `Session` metrics map, the metrics objects contained in this map are instances of the `IceMX::SessionMetrics` class show above. To configure a metrics view to record Glacier2 session metrics you can use metrics properties with the `IceMX.Metrics.`*`view-name`*`.Map.Session` prefix, for example:

- `IceMX.Metrics.SessionView.Map.Session.GroupBy=id` to configure a view containing one metrics object per session.
- `IceMX.Metrics.SessionView.Map.Session.GroupBy=none` to configure a view containing a single metrics object with metrics for all the sessions.

You can use the following attributes when configuring the Glacier2 Session metrics map:

Name	Description
id	A unique identifier to identify the session. This corresponds to the user Id for sessions created with a username/password and to the subject DN for sessions created from SSL.
parent	The Glacier2 router instance name.
none	The empty string.
endpoint	The stringified endpoint.
endpointType	The endpoint numerical type as defined in `Ice/Endpoint.ice`.
endpointIsDatagram	A boolean indicating if the endpoint is a datagram endpoint.
endpointIsSecture	A boolean indicating if the endpoint is secure.
endpointTimeout	The endpoint timeout.
endpointCompress	A boolean indicating if the endpoint requires compression.
endpointHost	The endpoint host.
endpointPort	The endpoint port.
connection	The connection description.
incoming	A boolean indicating if the connection is a server or client connection.

adapterName	If the connection is a server connection, adapterName will return the name of the adapter which created the connection, it will contain the empty string otherwise.
connectionId	The ID of the connection if one is set, the empty string otherwise.
localAddress	The connection's local address.
localPort	The connection's local port.
remoteAddress	The connection's remote address.
remotePort	The connection's remote port.
mcastAddress	The connection's multicast address.
mcastPort	The connection's multicast port.
state	The state of the connection.

The connection and endpoint attributes are for the connection tied to the Glacier2 session.

See Also

- Administrative Facility
- The Metrics Facet
- IceMX.Metrics.*

IceBridge

IceBridge is an Ice service that acts as a bridge between one or more clients and a server.

On this page:

IceBridge Overview

IceBridge relays requests from clients to a *target server* and makes every effort to be as transparent as possible. One example use case for IceBridge is when a client needs to communicate with a server over a particular transport, but the client machine doesn't support that transport. In this situation, an instance of IceBridge can be started on a host that does support the server's transport, and the client can use the bridge as an intermediary to reach the server.

IceBridge provides several features:

- **Connection matching**
  For each incoming connection from a client, the bridge creates a corresponding connection to the server. Furthermore, the lifetimes of the two connections are bound together. If you use ACM heartbeats to keep connections open, the bridge will automatically relay heartbeats in either direction. If one of the connections closes, the bridge will close its corresponding connection. These features make IceBridge usable for session-based applications, where application-specific semantics are usually associated with connections.

- **Transport matching**
  When IceBridge creates a connection to the server, it attempts to match the characteristics of the incoming connection from the client. For example, if the bridge receives a request from the client over a datagram transport, it will attempt to forward the request as a datagram. Similarly, if a client request arrives over a secure connection, the bridge will attempt to create a secure connection to the server.

- **Bidirectional requests**
  IceBridge configures every connection to the server to support bidirectional requests. All bidirectional callback requests sent from the server are automatically forwarded back to the client via the client's connection with the bridge.

- **Router support**
  IceBridge implements the `Ice::Router` interface so that it can be easily configured into a client as an Ice router. For applications with more complex router requirements, we recommend using Glacier2.

The next section describes how to configure IceBridge.

Configuring IceBridge

IceBridge supports the following properties:

- `IceBridge.Source.Endpoints`
  This required property lists the endpoints on which IceBridge listens for connections from clients. `IceBridge.Source` is also the name of an object adapter, which means all of the other object adapter properties can be configured as well.

- `IceBridge.Target.Endpoints`
  This required property identifies the endpoints of the target server. Note that listing multiple endpoints in this property means IceBridge will follow the usual Ice process for establishing a connection to the server. However, once IceBridge has established a matching connection, it will continue to use that connection for the lifetime of the client's connection to the bridge.

- `IceBridge.InstanceName`
  This optional property specifies a default identity category for the IceBridge objects. If not specified, the default value is `IceBridge`.

You will also need to configure IceBridge to load any transport plug-ins required by either the source or target endpoints.

Here's a simple example:

```
IceBridge.Source.Endpoints=tcp -p 10000
IceBridge.Target.Endpoints=tcp -h target.host -p 21112
```

The bridge listens on TCP port 10000 connections from clients, and forwards requests to the target server on TCP port 21112.

It's important to give some thought to your source and target endpoint configurations, also taking into consideration IceBridge's transport matching behavior that we described earlier. Consider this example:

```
IceBridge.Source.Endpoints=udp -p 10000
IceBridge.Target.Endpoints=tcp -h target.host -p 21112
```

This configuration will fail: when the bridge receives a datagram request from the client on its source endpoint, it will attempt to forward it as a datagram to the server. However, the target configuration only defines a TCP endpoint, which means the bridge's forwarding attempt cannot succeed. Here's another failure scenario:

```
IceBridge.Source.Endpoints=ssl -p 10000
IceBridge.Target.Endpoints=tcp -h target.host -p 21112
```

IceBridge will accept a secure connection from a client and will attempt to establish a secure connection to the target server, but the target configuration does not include a secure endpoint. Reversing the transports produces a working configuration:

```
IceBridge.Source.Endpoints=tcp -p 10000
IceBridge.Target.Endpoints=ssl -h target.host -p 21112
```

This configuration succeeds because an SSL connection to the target is compatible with a TCP connection from a client.

Generally speaking, your source endpoints need to accommodate the client's requirements, and the target endpoints need to provide compatible transports for the source endpoints. The example below shows how to successfully offer multiple transports:

```
IceBridge.Source.Endpoints=tcp -p 10000:udp -p 10000
IceBridge.Target.Endpoints=tcp -h target.host -p 21112:udp -p
target.host -p 40444
```

This configuration allows a client to use both connection-oriented (TCP) and connectionless (UDP) transports when communicating with the target server via the bridge.

One last example demonstrates how to bridge between TCP and Bluetooth using the IceBT transport plug-in:

```
IceBridge.Source.Endpoints=tcp -p 10000
IceBridge.Target.Endpoints=bt -a "01:23:45:67:89:AB" -u
"6a193943-1754-4869-8d0a-ddc5f9a2b294"
```

With this configuration, a client can connect to the bridge using TCP, and the bridge will establish a Bluetooth connection to the device with the given address offering the service identified by the given UUID.

IceBridge Object Identities

An IceBridge server hosts one well-known object. The default identity of this object is `IceBridge/router`, corresponding to the `Ice::Router` interface.

Clients can configure a router proxy using this identity together with the bridge's source endpoints. This object identity is reserved for use by the bridge, therefore any client requests having this identity will be dispatched to the internal router object and not forwarded to the target. If the application requires a different identity, you can set the `IceBridge.InstanceName` property to change the category of the object identity as shown in the example below:

```
IceBridge.InstanceName=PublicBridge
```

This property changes the category of the object identity, which becomes `PublicBridge/router`. The client's configuration must also be changed to reflect the new identity:

```
Ice.Default.Router=PublicBridge/router:tcp -h 5.6.7.8 -p 4063
```

A client can discover the bridge's proxy for its router at run time using the `RouterFinder` interface.

Using IceBridge

Clients will require configuration changes to use IceBridge but shouldn't normally require any code changes. The first step is evaluating whether your client should use IceBridge as a router:

- Does the target server create and return proxies that the client uses for subsequent invocations? If so, you must configure the client to use IceBridge as a router. Doing so forces the Ice run time in the client to ignore the endpoints that the server returned in these proxies and use the IceBridge endpoints instead.
- Does the client statically configure a number of proxies? If so, configuring IceBridge as a router is convenient but not mandatory. Again, using IceBridge as a router causes Ice to ignore the endpoints in arbitrary proxies and instead use the bridge endpoints. This avoids having to manually modify all of the statically-configured proxies to use the IceBridge source endpoints.
- Otherwise, you can either configure your client to use IceBridge as a router, or modify your client's proxies to have the IceBridge source endpoints. We provide examples of both scenarios below.

Let's assume the bridge has the following configuration:

Bridge Configuration

```
IceBridge.Target.Endpoints=...
IceBridge.Source.Endpoints=tcp -p 10000
```

The client can use IceBridge as a router by defining `Ice.Default.Router`:

Client Configuration with Router

```
Ice.Default.Router=IceBridge/router:tcp -h bridge.host -p 10000
Client.Proxy=SomeObject:tcp -h other.host -p 9999
```

This configuration causes the Ice run time in the client to ignore the endpoint in `Client.Proxy` and instead send all requests via the given router.

Setting `Ice.Default.Router` affects **all** proxies by default. Ice also provides more selective ways of configuring a router, such as with a proxy property or a proxy method.

If you've decided not to use IceBridge as a router, you simply need to replace the existing endpoints in the client's proxies with the bridge's source endpoints:

Client Configuration without Router

```
Client.Proxy=SomeObject:tcp -h bridge.host -p 10000
```

Starting IceBridge

The bridge supports the following command-line options:

```
$ icebridge -h
Usage: icebridge [options]
Options:
-h, --help     Show this message.
-v, --version  Display the Ice version.
```

Additional command line options are supported, including those that allow the router to run as a Windows service or Unix daemon.

Assuming our configuration properties are stored in a file named `config`, you can start the bridge with the following command:

```
$ icebridge --Ice.Config=config
```

IceBridge Limitations

Although IceBridge attempts to be as transparent as possible, it does have some limitations that you should be aware of.

Single target server

A single IceBridge instance can support multiple clients simultaneously, however the requests are being forwarded to a single target server. Each connection from a client results in the bridge creating a corresponding connection to the target server, but all of the clients of a bridge are logically connecting to the same target. While it's true that the bridge can be configured with multiple target endpoints, the bridge simply treats them as multiple options for connecting to the same server.

If your clients need to bridge to multiple servers, you must start a separate IceBridge instance for each target server.

Clients that create object adapters

A *mixed client-server* application is one that creates an object adapter in order to receive new incoming connections, while a *bidirectional client* creates an object adapter solely to receive callbacks over an existing outgoing connection that has been configured for bidirectional requests. IceBridge supports bidirectional clients, but a mixed client-server application may require more administrative effort. For example, if the application wants to use IceBridge for its outgoing connections, and also use IceBridge for its incoming connections, then you would need to start two instances of IceBridge, one for each direction.

SSL credentials

IceBridge can act as a secure "man in the middle", but only using a single set of credentials. In other words, the identity you configure for

IceSSL will be used to accept secure incoming connections from clients, and to establish secure outgoing connections to the target server. IceBridge currently does not provide the ability to configure separate identities for each of these activities.

Bluetooth connection limit

As mentioned in the IceBT discussion, a Bluetooth client process cannot establish multiple connections to the same target endpoint. When using IceBridge with a Bluetooth target, only one client at a time can use the bridge. Furthermore, that client must only establish one connection to the bridge. You can start additional IceBridge instances to allow more clients to communicate with the Bluetooth device simultaneously.

Session support

Session-based applications that assign semantics to connections can use IceBridge because it maintains a one-to-one relationship between incoming connections from clients and outgoing connections to the target. However, IceBridge provides no support for session authentication or authorization. If your application requires these features, we recommend using Glacier2 instead.

Connection closure

If one of the bridge's connections closes, the bridge immediately closes its matching connection. The remote end of this connection will currently detect this as a dropped connection, rather than as a connection that was closed gracefully.

IceGrid

IceGrid is a location and activation service for Ice applications. For the purposes of this discussion, we can loosely define *grid computing* as the use of a network of relatively inexpensive computers to perform the computational tasks that once required costly "big iron." Developers familiar with distributed computing technologies may not consider the notion of grid computing to be particularly revolutionary; after all, distributed applications have been running on networks for years, and the definition of grid computing is sufficiently vague that practically any server environment could be considered a "grid."

One possible grid configuration is a homogeneous collection of computers running identical programs. Each computer in the grid is essentially a clone of the others, and all are equally capable of handling a task. As a developer, you need to write the code that runs on the grid computers, and Ice is ideally suited as the infrastructure that enables the components of a grid application to communicate with one another. However, writing the application code is just the first piece of the puzzle. Many other challenges remain:

- How do I install and update this application on all of the computers in the grid?
- How do I keep track of the servers running on the grid?
- How do I distribute the load across all the computers?
- How do I migrate a server from one computer to another one?
- How can I quickly add a new computer to the grid?

Of course, these are issues faced by most distributed applications. As you learn more about IceGrid's capabilities, you will discover that it offers solutions to these challenges. To get you started, we have summarized IceGrid's feature set below:

- Location service
  As an implementation of an Ice location service, IceGrid enables clients to bind indirectly to their servers, making applications more flexible and resilient to changing requirements. (IceDiscovery is an alternate, lightweight location service implementation for applications that don't require IceGrid's additional features.)

- On-demand server activation
  Starting an Ice server process is called *server activation*. IceGrid can be given responsibility for activating a server on demand, that is, when a client attempts to access an object hosted by the server. Activation usually occurs as a side effect of indirect binding, and is completely transparent to the client.

- Application distribution
  IceGrid provides a convenient way to distribute your application to a set of computers, without the need for a shared file system or complicated scripts. Simply configure an IcePatch2 server and let IceGrid download the necessary files and keep them synchronized.

- Replication and load balancing
  IceGrid supports replication by grouping the object adapters of several servers into a single virtual object adapter. During indirect binding, a client can be bound to an endpoint of any of these adapters. Furthermore, IceGrid monitors the load on each computer and can use that information to decide which of the endpoints to return to a client.

- Sessions and resource allocation
  An IceGrid client establishes a session in order to allocate a resource such as an object or a server. IceGrid prevents other clients from using the resource until the client releases it or the session expires. Sessions enhance security through the use of an authentication mechanism that can be integrated with a Glacier2 router.

- Automatic failover
  Ice supports automatic retry and failover in any proxy that contains multiple endpoints. When combined with IceGrid's support for replication and load balancing, automatic failover means that a failed request results in a client transparently retrying the request on the next endpoint with the lowest load.

- Dynamic queries
  In addition to transparent binding, applications can interact directly with IceGrid to locate objects in a variety of ways.

- Status monitoring
  IceGrid supports Slice interfaces that allow applications to monitor its activities and receive notifications about significant events, enabling the development of custom tools or the integration of IceGrid status events into an existing management framework.

- Administration
  IceGrid includes command-line and graphical administration tools. They are available on all supported platforms and allow you to start, stop, monitor, and reconfigure any server managed by IceGrid.

- Deployment
  Using XML files, you can describe the servers to be deployed on each computer. Templates simplify the description of identical servers.

As grid computing enters the mainstream and compute servers become commodities, users expect more value from their applications. IceGrid, in cooperation with the Ice run time, relieves you of these low-level tasks to accelerate the construction and simplify the administration of your distributed applications.

Topics

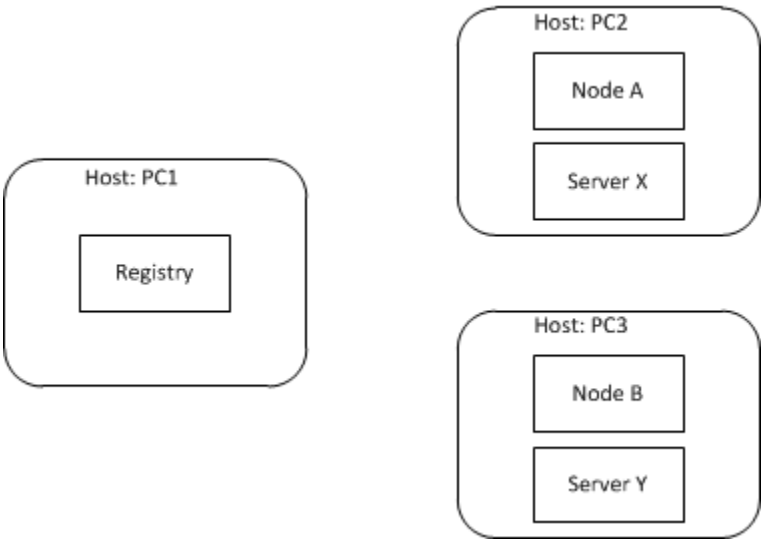
IceGrid Architecture

An IceGrid domain consists of a *registry* and any number of *nodes*. Together, the registry and nodes cooperate to manage the information and server processes that comprise *applications*. Each application assigns servers to particular nodes. The registry maintains a persistent record of this information, while the nodes are responsible for starting and monitoring their assigned server processes. In a typical configuration, one node runs on each computer that hosts Ice servers. The registry does not consume much processor time, so it commonly runs on the same computer as a node; in fact, the registry and a node can run in the same process if desired. If fault tolerance is desired, the registry supports replication using a master-slave design.

On this page:

Architecture of a Simple IceGrid Application

As an example, this illustration shows a very simple IceGrid application running on a network of three computers. The IceGrid registry is the only process of interest on host `PC1`, while IceGrid nodes are running on the hosts `PC2` and `PC3`. In this sample application, one server has been assigned to each node.



Simple IceGrid application.

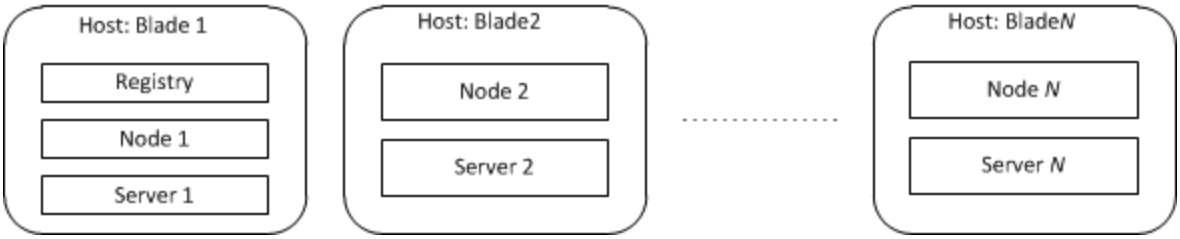
From a client application's perspective, the primary responsibility of the registry is to resolve indirect proxies as an Ice location service. As such, this contribution is largely transparent: when a client first attempts to use an indirect proxy, the Ice run time in the client contacts the registry to convert the proxy's symbolic information into endpoints that allow the client to establish a connection.

Although the registry might sound like nothing more than a simple lookup table, reality is quite different. For example, behind the scenes, a locate request might prompt a node to start the target server automatically, or the registry might select appropriate endpoints based on load statistics from each computer.

This also illustrates the benefits of indirect proxies: the location service can provide a great deal of functionality without any special action by the client and, unlike with direct proxies, the client does not need advance knowledge of the address and port of a server. The extra level of indirection adds some latency to the client's first use of a proxy; however, all subsequent interactions occur directly between client and server, so the cost is negligible. Furthermore, indirection allows servers to migrate to different computers without the need to update proxies held by clients.

Server Replication with IceGrid

IceGrid's flexibility allows an endless variety of configurations. For example, suppose we have a grid network and want to replicate a server on each blade, as shown below:



Replicated server on grid network.

Replication in Ice is based on object adapters, not servers. Any object adapter in any server could participate in replication, but it is far more likely that all of the replicated object adapters are created by instances of the same server executable that is running on each computer. We are using this configuration in the example shown above, but IceGrid requires each server to have a unique name. `Server 1` and `Server 2` are our unique names for the same executable.

The binding process works somewhat differently when replication is involved, since the registry now has multiple object adapters to choose from. The description of the IceGrid application drives the registry's decision about which object adapter (or object adapters) to use. For example, the registry could consider the system load of each computer (as periodically reported by the nodes) and return the endpoints of the object adapter on the computer with the lowest load. It is also possible for the registry to combine the endpoints of several object adapters, in which case the Ice run time in the client would select the endpoint for the initial connection attempt.

Deploying an IceGrid Application

In IceGrid, *deployment* is the process of describing an application to the registry. This description includes the following information:

- Replica groups
  A *replica group* is the term for a collection of replicated object adapters. An application can create any number of replica groups. Each group requires a unique identifier.
- Nodes
  An application must assign its servers to one or more nodes.
- Servers
  A server's description includes a unique name and the path to its executable. It also lists the object adapters it creates.
- Object adapters
  Information about an object adapter includes its endpoints and any well-known objects it advertises. If the object adapter is a member of a replica group, it must also supply that group's identifier.
- Objects
  A *well-known object* is one that is known solely by its identity. The registry maintains a global list of such objects for use during locate requests.

IceGrid uses the term *descriptor* to refer to the description of an application and its components; deploying an application involves creating its descriptors in the registry. The are several ways to accomplish this:

- You can use a command-line tool that reads a file containing an XML representation of the descriptors.
- You can create descriptors interactively with the graphical administration tool.
- You can create descriptors programmatically via IceGrid's administrative interface.

The registry server must be running in order to deploy an application, but it is not necessary for nodes to be active. Nodes that are started after deployment automatically retrieve the information they need from the registry. Once deployed, you can update the application at any time.

See Also

- Locators
- Connection Establishment
- Object Adapters
- Object Adapter Replication
- Well-Known Objects

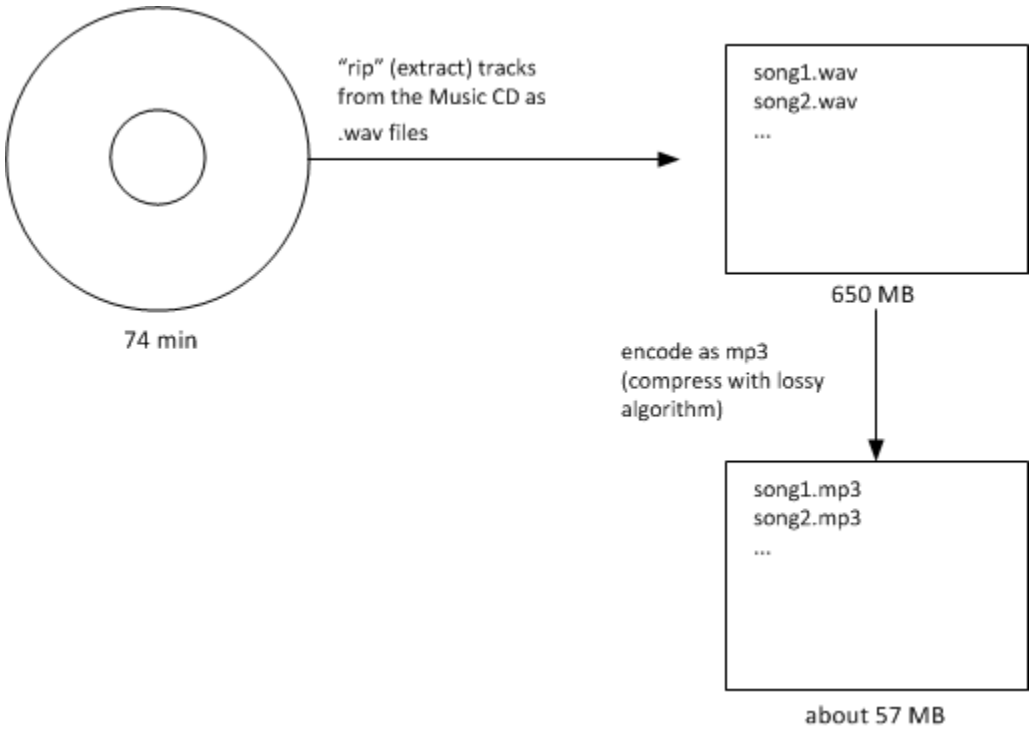
Getting Started with IceGrid

This page introduces a sample application that will help us demonstrate IceGrid's capabilities.

On this page:

The Ripper Application

Our application "rips" music tracks from a compact disc (CD) and encodes them as MP3 files, as shown below:



Overview of sample application.

Ripping an entire CD usually takes several minutes because the MP3 encoding requires lots of CPU cycles. Our distributed ripper application accelerates this process by taking advantage of powerful CPUs on remote Ice servers, enabling us to process many songs in parallel.

The Slice interface for the MP3 encoder is straightforward:

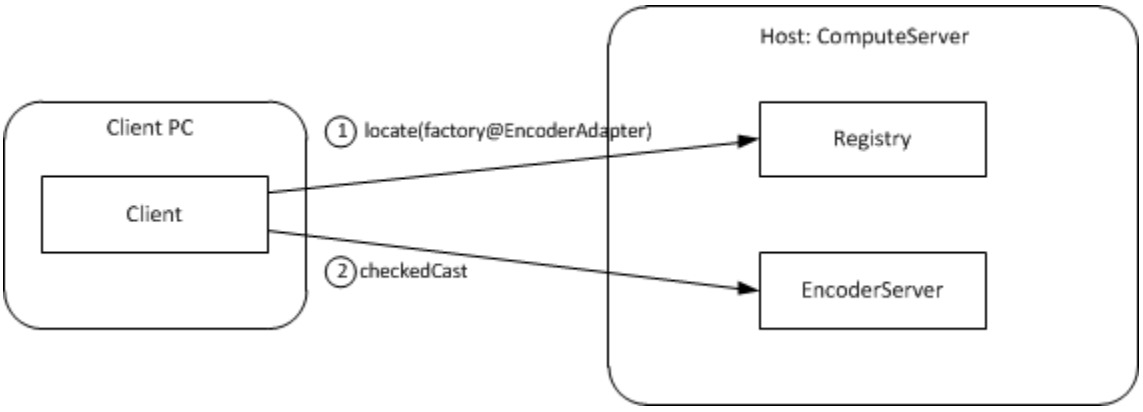
Slice

```
#include <Ice/BuiltinSequences.ice>
module Ripper
{
    exception EncodingFailedException
    {
        string reason;
    }

    sequence<short> Samples;

    interface Mp3Encoder
    {
        // Input: PCM samples for left and right channels
        // Output: MP3 frame(s).
        Ice::ByteSeq encode(Samples leftSamples, Samples rightSamples)
            throws EncodingFailedException;

        // You must flush to get the last frame(s). Flush also
        // destroys the encoder object.
        Ice::ByteSeq flush() throws EncodingFailedException;
    }

    interface Mp3EncoderFactory
    {
        Mp3Encoder* createEncoder();
    }
}
```

The implementation of the encoding algorithm is not relevant for the purposes of this discussion. Instead, we will focus on incrementally improving the application as we discuss IceGrid features.

Initial Ripper Architecture

The initial architecture for our application is intentionally simple, consisting of an IceGrid registry and a server that we start manually. This illustration shows how the client's invocation on its `EncoderFactory` proxy causes an implicit locate request:



Initial architecture for the ripper application.

The corresponding C++ code for the client is presented below:

C++11 C++98

```
auto proxy = communicator->stringToProxy("factory@EncoderAdapter");
auto factory = Ice::checkedCast<Ripper::MP3EncoderFactoryPrx>(proxy);
auto encoder = factory->createEncoder();
```

```
Ice::ObjectPrx proxy =
communicator->stringToProxy("factory@EncoderAdapter");
Ripper::MP3EncoderFactoryPrx factory =
Ripper::MP3EncoderFactoryPrx::checkedCast(proxy);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

Notice that the client uses an indirect proxy for the `MP3EncoderFactory` object. This stringified proxy can be read literally as "the object with identity `factory` in the object adapter identified as `EncoderAdapter`." The encoding server creates this object adapter and ensures that the object adapter uses this identifier. Since each object adapter must be uniquely identified, the registry can easily determine the server that created the adapter and return an appropriate endpoint to the client.

The client's call to `checkedCast` is the first remote invocation on the factory object, and therefore the locate request is performed during the completion of this invocation. The subsequent call to `createEncoder` is sent directly to the server without further involvement by IceGrid.

Ripper Registry Configuration

The registry needs a subdirectory in which to create its databases, and we will use `/opt/ripper/registry` for this purpose (the directory must exist before starting the registry). We also need to create an Ice configuration file to hold properties required by the registry. The file `/opt/ripper/registry.cfg` contains the following properties:

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifi
er
IceGrid.Registry.LMDB.Path=/opt/ripper/registry
IceGrid.Registry.DynamicRegistration=1
```

Several of the properties define endpoints, but only the value of `IceGrid.Registry.Client.Endpoints` needs a fixed port. This property specifies the endpoints of the IceGrid locator service; IceGrid clients must include these endpoints in their definition of `Ice.Default.Locator`, as discussed in the next section. The TCP port number (4061) used in this example has been reserved by the Internet Assigned Numbers Authority (IANA) for the IceGrid registry, along with SSL port number 4062.

Several other properties are worth mentioning:

- `IceGrid.Registry.AdminPermissionsVerifier`
  Controls access to the registry's administrative functionality.
- `IceGrid.Registry.LMDB.Path`
  Specifies the registry's database directory.
- `IceGrid.Registry.DynamicRegistration`
  If set to a non-zero value, allows servers to register their object adapters. Dynamic registration is explained in more detail below.

By default, IceGrid will not permit a server to register its object adapters without using IceGrid's deployment facility. In some situations, such as in this sample application, you may want a client to be able to bind indirectly to a server without having to first deploy the server. That is, simply starting the server should be sufficient to make the server register itself with IceGrid and be reachable from clients.

You can achieve this by running the registry with the property `IceGrid.Registry.DynamicRegistration` set to a non-zero value. With this setting, IceGrid permits an adapter to register itself upon activation even if it has not been previously deployed. To force the server to register its adapters, you must define `Ice.Default.Locator` (so the server can find the registry) and, for each adapter that you wish to register, you must set `<adapter-name>.AdapterId` to an identifier that is unique within the registry. Setting the `<adapter-name>.AdapterId` property also causes the adapter to no longer create direct proxies but rather to create indirect proxies that clients must resolve via the registry.

Ripper Client Configuration

The client requires only minimal configuration, namely a value for the property `Ice.Default.Locator`. This property supplies the Ice run time with the proxy for the locator service. In IceGrid, the locator service is implemented by the registry, and the locator object is available on the registry's client endpoints. The property `IceGrid.Registry.Client.Endpoints` defined above provides most of the information we need to construct the proxy. The missing piece is the identity of the locator object, which defaults to `IceGrid/Locator` but may change based on the registry's configuration:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

The use of a locator service allows the client to take advantage of indirect binding and avoid static dependencies on server endpoints. However, the locator proxy must have a fixed port, otherwise the client has a bootstrapping problem: it cannot resolve indirect proxies without knowing the endpoints of the locator service.

> IceLocatorDiscovery eliminates the need for a client to define the `Ice.Default.Locator` property by using UDP multicast to discover registries at run time.

Ripper Server Configuration

We use `/opt/ripper/server.cfg` as the server's configuration file. It contains the following properties:

```
EncoderAdapter.AdapterId=EncoderAdapter
EncoderAdapter.Endpoints=tcp
Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

The properties are described below:

- `EncoderAdapter.AdapterId`
  This property supplies the object adapter identifier that the client uses in its indirect proxy (e.g., `factory@EncoderAdapter`).
- `EncoderAdapter.Endpoints`
  This property defines the object adapter's endpoint. Notice that the value does not contain any port information, meaning that the adapter uses a system-assigned port. Without IceGrid, the use of a system-assigned port would pose a significant problem: how would a client create a direct proxy if the adapter's port could change every time the server is restarted? IceGrid solves this problem nicely because clients can use indirect proxies that contain no endpoint dependencies. The registry resolves indirect proxies using the endpoint information supplied by object adapters each time they are activated.
- `Ice.Default.Locator`
  The server requires a value for this property in order to register its object adapter.

Starting the Registry for the Ripper Application

Now that the configuration file is written and the directory structure is prepared, we are ready to start the IceGrid registry:

```
$ icegridregistry --Ice.Config=/opt/ripper/registry.cfg
```

Additional command line options are supported, including those that allow the registry to run as a Windows service or Unix daemon.

Starting the Ripper Server

With the registry up and running, we can now start the server. At a command prompt, we run the program and pass an `--Ice.Config` option indicating the location of the configuration file:

```
$ /opt/ripper/bin/server --Ice.Config=/opt/ripper/server.cfg
```

Ripper Progress Review

This example demonstrated how to use IceGrid's location service, which is a core component of IceGrid's feature set. By incorporating IceGrid into our application, the client is now able to locate the `MP3EncoderFactory` object using only an indirect proxy and a value for `Ice.Default.Locator`. Furthermore, we can reconfigure the application in any number of ways without modifying the client's code or configuration.

For some applications, the functionality we have already achieved using IceGrid may be entirely sufficient. However, we have only just begun to explore IceGrid's capabilities, and there is much we can still do to improve our application. The next section shows how we can avoid the need to start our server manually by deploying our application onto an IceGrid node.

See Also

- Locator Configuration for a Client
- Resource Allocation using IceGrid Sessions
- Well-Known Registry Objects
- Using IceGrid Deployment
- Object Adapter Endpoints
- icegridregistry
- IceGrid.*

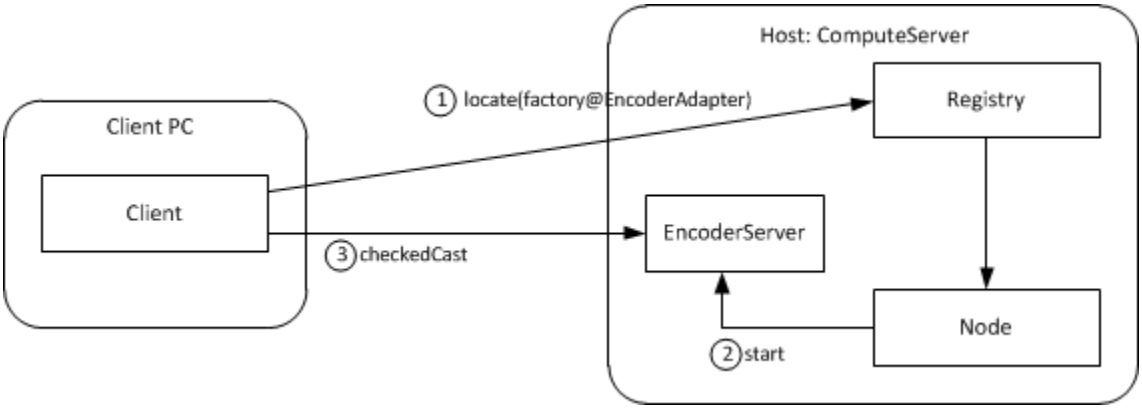
Using IceGrid Deployment

Here we extend the capabilities of our sample application using IceGrid's deployment facility.

On this page:

Ripper Architecture using Deployment

The revised architecture for our application consists of a single IceGrid node responsible for our encoding server that runs on the computer named `ComputeServer`. The illustration below shows the client's initial invocation on its indirect proxy and the actions that IceGrid takes to make this invocation possible:



Architecture for deployed ripper application.

In contrast to the initial architecture, we no longer need to manually start our server. In this revised application, the client's locate request prompts the registry to query the node about the server's state and start it if necessary. Once the server starts successfully, the locate request completes and subsequent client communication occurs directly with the server.

Ripper Deployment Descriptors

We can deploy our application using the `icegridadmin` command line utility, but first we must define our descriptors in XML. The descriptors are quite brief:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" id="EncoderAdapter"
endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

For IceGrid's purposes, we have named our application `Ripper`. It consists of a single server, `EncoderServer`, assigned to the node `Node 1`.

> Since a computer typically runs only one node process, you might be tempted to give the node a name that identifies its host (such as `ComputeServerNode`). However, this naming convention becomes problematic as soon as you need to migrate the node to another host.

The server's `exe` attribute supplies the pathname of its executable, and the `activation` attribute indicates that the server should be activated on demand when necessary.

The object adapter's descriptor is the most interesting. As you can see, the `name` and `id` attributes both specify the value `EncoderAdapter`. The value of `name` reflects the adapter's name in the server process (i.e., the argument passed to `createObjectAdapter`) that is used for configuration purposes, whereas the value of `id` uniquely identifies the adapter within the registry and is used in indirect proxies. These attributes are not required to have the same value. Had we omitted the `id` attribute, IceGrid would have composed a unique value by combining the server name and adapter name to produce the following identifier:

```
EncoderServer.EncoderAdapter
```

The `endpoints` attribute defines one or more endpoints for the adapter. As explained earlier, these endpoints do not require a fixed port.

Refer to the XML reference for detailed information on using XML to define descriptors.

Ripper Registry and Node Configuration

In our initial registry configuration, we created the directory `/opt/ripper/registry` for use by the registry. The node also needs a subdirectory for its own purposes, so we will use `/opt/ripper/node`. Again, these directories must exist before starting the registry and node.

We also need to create an Ice configuration file to hold properties required by the registry and node. The file `/opt/ripper/config` contains the following properties:

```
# Registry properties
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifi
er
IceGrid.Registry.LMDB.Path=/opt/ripper/registry

# Node properties
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node1
IceGrid.Node.Data=/opt/ripper/node
IceGrid.Node.CollocateRegistry=1
Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

The registry and node can share this configuration file. In fact, by enabling `IceGrid.Node.CollocateRegistry`, we have indicated that the registry and node should run in the same process.

One difference from our initial configuration is that we no longer define `IceGrid.Registry.DynamicRegistration`. By omitting this property, we force the registry to reject the registration of object adapters that have not been deployed.

The node properties are explained below:

- `IceGrid.Node.Endpoints`
  This property specifies the node's endpoints. A fixed port is not required.
- `IceGrid.Node.Name`
  This property defines the unique name for this node. Its value must match the descriptor we wrote above.
- `IceGrid.Node.Data`
  This property specifies the node's data directory.
- `Ice.Default.Locator`
  This property is defined for use by the `icegridadmin` tool. The node would also require this property if the registry is not collocated. Refer to our discussion of the ripper client configuration for more information on this setting.

Ripper Server Configuration using Deployment

Server configuration is accomplished using descriptors. During deployment, the node creates a subdirectory tree for each server. Inside this tree the node creates a configuration file containing properties derived from the server's descriptors. For instance, the adapter's descriptor generates the following properties in the server's configuration file:

```
# Server configuration
Ice.Admin.ServerId=EncoderServer
Ice.Admin.Endpoints=tcp -h 127.0.0.1
Ice.ProgramName=EncoderServer
# Object adapter EncoderAdapter
EncoderAdapter.Endpoints=tcp
EncoderAdapter.AdapterId=EncoderAdapter
Ice.Default.Locator=IceGrid/Locator:default -p 4061
```

As you can see, the configuration file that IceGrid generates from the descriptor resembles the initial configuration, with two additional properties:

- `Ice.Admin.ServerId`
- `Ice.Admin.Endpoints`

The `Ice.Admin.Endpoints` property enables the administrative facility that, among other features, allows an IceGrid node to gracefully deactivate the server.

Using the directory structure we established for our ripper application, the configuration file for `EncoderServer` has the file name shown below:

```
/opt/ripper/node/servers/EncoderServer/config/config
```

Note that this file should not be edited directly because any changes you make are lost the next time the node regenerates the file. The correct way to add properties to the file is to include property definitions in the server's descriptor. For example, we can add the property `Ice.Trace.Network=1` by modifying the server descriptor as follows:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" id="EncoderAdapter"
endpoints="tcp"/>
                <property name="Ice.Trace.Network" value="1"/>
            </server>
        </node>
    </application>
</icegrid>
```

When a node activates a server, it passes the location of the server's configuration file using the `--Ice.Config` command-line argument. If you start a server manually from a command prompt, you must supply this argument yourself.

Starting the Node for the Ripper Application

Now that the configuration file is written and the directory structure is prepared, we are ready to start the IceGrid registry and node. Using a collocated registry and node, we only need to use one command:

```
$ icegridnode --Ice.Config=/opt/ripper/config
```

Additional command line options are supported, including those that allow the node to run as a Windows service or Unix daemon.

Deploying the Ripper Application

With the registry up and running, it is now time to deploy our application. Like our client, the `icegridadmin` utility also requires a definition for the `Ice.Default.Locator` property. We can start the utility with the following command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
```

After confirming that it can contact the registry, `icegridadmin` provides a command prompt at which we deploy our application. Assuming our descriptor is stored in `/opt/ripper/app.xml`, the deployment command is shown below:

```
>>> application add "/opt/ripper/app.xml"
```

Next, confirm that the application has been deployed:

```
>>> application list
Ripper
```

You can start the server using this command:

```
>>> server start EncoderServer
```

Finally, you can retrieve the current endpoints of the object adapter:

```
>>> adapter endpoints EncoderAdapter
```

If you want to experiment further using `icegridadmin`, issue the `help` command and review the available commands.

Ripper Progress Review

We have deployed our first IceGrid application, but you might be questioning whether it was worth the effort. Even at this early stage, we have already gained several benefits:

- We no longer need to manually start the encoder server before starting the client, because the IceGrid node automatically starts it if it is not active at the time a client needs it. If the server happens to terminate for any reason, such as an IceGrid administrative action or a server programming error, the node restarts it without intervention on our part.
- We can manage the application remotely using one of the IceGrid administration tools. The ability to remotely modify applications, start and stop servers, and inspect every aspect of your configuration is a significant advantage.

Admittedly, we have not made much progress yet in our stated goal of improving the performance of the ripper over alternative solutions that are restricted to running on a single computer. Our client now has the ability to easily delegate the encoding task to a server running on another computer, but we have not achieved the parallelism that we really need. For example, if the client created a number of encoders and used them simultaneously from multiple threads, the encoding performance might actually be *worse* than simply encoding the data directly in the client, as the remote computer would likely slow to a crawl while attempting to task-switch among a number of processor-intensive tasks.

Adding Nodes to the Ripper Application

Adding more nodes to our environment would allow us to distribute the encoding load to more compute servers. Using the techniques we have learned so far, let us investigate the impact that adding a node would have on our descriptors, configuration, and client application.

Descriptor Changes

The addition of a node is mainly an exercise in cut and paste:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

Note that we now have two `node` elements instead of a single one. You might be tempted to simply use the host name as the node name. However, in general, that is not a good idea. For example, you may want to run several IceGrid nodes on a single machine (for example, for testing). Similarly, you may have to rename a host at some point, or need to migrate a node to a different host. But, unless you also rename the node, that leads to the situation where you have a node with the name of a (possibly obsolete) host when the node in fact is not running on that host. Obviously, this makes for a confusing configuration — it is better to use abstract node names, such as `Node1`.

Aside from the new `node` element, notice that the server identifiers must be unique. The adapter name, however, can remain as `EncoderAdapter` because this name is used only for local purposes within the server process. In fact, using a different name for each adapter would actually complicate the server implementation, since it would somehow need to discover the name it should use when creating the adapter.

We have also removed the `id` attribute from our adapter descriptors; the default values supplied by IceGrid are sufficient for our purposes.

Configuration Changes

We can continue to use the configuration file we created earlier for our combined registry-node process. We need a separate configuration file for `Node2`, primarily to define a different value for the property `IceGrid.Node.Name`. However, we also cannot have two nodes configured with `IceGrid.Node.CollocateRegistry` because only one master registry is allowed, so we must remove this property:

```
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node2
IceGrid.Node.Data=/opt/ripper/node

Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

We assume that `/opt/ripper/node` refers to a local file system directory on the computer hosting `Node2`, and not a shared volume, because two nodes must not share the same data directory.

We have also modified the locator proxy to include the address of the host on which the registry is running.

> Using IceLocatorDiscovery allows a node to discover its registry at run time without the need to define `Ice.Default.Locator`.

Redeploying the Application

After saving the new descriptors, you need to redeploy the application. Using `icegridadmin`, issue the following command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> application update "/opt/ripper/app.xml"
```

If an update affects any of the application's servers that are currently running, IceGrid automatically stops those servers prior to performing the update and restarts them again after the update is complete. We can determine whether an update would require any restarts using the `application diff` command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> application diff --servers "/opt/ripper/app.xml"
```

To ensure that our update does not impact any active servers, we can use the `--no-restart` option:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> application update --no-restart "/opt/ripper/app.xml"
```

With this option, the update would fail if any servers required a restart.

Client Changes

We have added a new node, but we still need to modify our client to take advantage of it. As it stands now, our client can delegate an encoding task to one of the two `MP3EncoderFactory` objects. The client selects a factory by using the appropriate indirect proxy:

- `factory@EncoderServer1.EncoderAdapter`
- `factory@EncoderServer2.EncoderAdapter`

In order to distribute the tasks among both factories, the client could use a random number generator to decide which factory receives the next task:

C++11 C++98

```
string adapter;
if((rand() % 2) == 0)
{
    adapter = "EncoderServer1.EncoderAdapter";
}
else
{
    adapter = "EncoderServer2.EncoderAdapter";
}
auto proxy = communicator->stringToProxy("factory@" + adapter);
auto factory = Ice::checkedCast<Ripper::MP3EncoderFactoryPrx>(proxy);
auto encoder = factory->createEncoder();
```

```
string adapter;
if((rand() % 2) == 0)
{
    adapter = "EncoderServer1.EncoderAdapter";
}
else
{
    adapter = "EncoderServer2.EncoderAdapter";
}
Ice::ObjectPrx proxy =
communicator->stringToProxy("factory@" + adapter);
Ripper::MP3EncoderFactoryPrx factory =
Ripper::MP3EncoderFactoryPrx::checkedCast(proxy);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

There are a few disadvantages in this design:

- The client application must be modified each time a new server is added or removed because it knows all of the adapter identifiers.
- The client cannot distribute the load intelligently; it is just as likely to assign a task to a heavily-loaded computer as it is an idle one.

We describe better solutions in the sections that follow.

See Also

- IceGrid Server Activation
- Creating an Object Adapter
- Object Adapter Endpoints
- Getting Started with IceGrid
- icegridadmin Command Line Tool
- IceGrid and the Administrative Facility
- icegridnode
- Adapter Descriptor Element
- IceGrid.*

Well-Known Objects

On this page:

Overview of Well-Known Objects

There are two types of indirect proxies: one specifies an identity and an object adapter identifier, while the other contains only an identity. The latter type of indirect proxy is known as a *well-known proxy*. A well-known proxy refers to a well-known object, that is, its identity alone is sufficient to allow the client to locate it. Ice requires all object identities in an application to be unique, but typically only a select few objects are able to be located only by their identities.

In earlier sections we showed the relationship between indirect proxies containing an object adapter identifier and the IceGrid configuration. Briefly, in order for a client to use a proxy such as `factory@EncoderAdapter`, an object adapter must be given the identifier `EncoderAdapter`.

A similar requirement exists for well-known objects. The registry maintains a table of these objects, which can be populated in a number of ways:

- statically in descriptors,
- programmatically using IceGrid's administrative interface,
- dynamically using an IceGrid administration tool.

The registry's database maps an object identity to a proxy. A locate request containing only an identity prompts the registry to consult this database. If a match is found, the registry examines the associated proxy to determine if additional work is necessary. For example, consider the well-known objects in the following table.

Identity	Proxy
`Object1`	`Object1:tcp -p 10001`
`Object2`	`Object2@TheAdapter`
`Object3`	`Object3`

The proxy associated with `Object1` already contains endpoints, so the registry can simply return this proxy to the client.

For `Object2`, the registry notices the adapter ID and checks to see whether it knows about an adapter identified as `TheAdapter`. If it does, it attempts to obtain the endpoints of that adapter, which may cause its server to be started. If the registry is successfully able to determine the adapter's endpoints, it returns a direct proxy containing those endpoints to the client. If the registry does not recognize `TheAdapter` or cannot obtain its endpoints, it returns the indirect proxy `Object2@TheAdapter` to the client. Upon receipt of another indirect proxy, the Ice run time in the client will try once more to resolve the proxy, but generally this will not succeed and the Ice run time in the client will raise a `NoEndpointException` as a result.

Finally, `Object3` represents a hopeless situation: how can the registry resolve `Object3` when its associated proxy refers to itself? In this case, the registry returns the proxy `Object3` to the client, which causes the client to raise `NoEndpointException`. Clearly, you should avoid this situation.

Well-Known Object Types

The registry's database not only associates an identity with a proxy, but also a type. Technically, the "type" is an arbitrary string but, by convention, that string represents the most-derived Slice type of the object. For example, the Slice type ID of the encoder factory in our ripper application is `::Ripper::MP3EncoderFactory`.

Object types are useful when performing queries.

Deploying Well-Known Objects

The `object` descriptor adds a well-known object to the registry. It must appear within the context of an adapter descriptor, as shown in the XML example below:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" id="EncoderAdapter"
endpoints="tcp">
                    <object identity="EncoderFactory"
type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

During deployment, the registry associates the identity `EncoderFactory` with the indirect proxy `EncoderFactory@EncoderAdapter`. If the adapter descriptor had omitted the adapter ID, the registry would have generated a unique identifier by combining the server ID and the adapter name.

In this example, the object's type is specified explicitly.

Adding Well-Known Objects Programmatically

The `IceGrid::Admin` interface defines several operations that manipulate the registry's database of well-known objects:

Slice

```
module IceGrid
{
    interface Admin
    {
        ...
        void addObject(Object* obj)
            throws ObjectExistsException,
               DeploymentException;
        void updateObject(Object* obj)
            throws ObjectNotRegisteredException,
               DeploymentException;
        void addObjectWithType(Object* obj, string type)
            throws ObjectExistsException,
               DeploymentException;
        void removeObject(Ice::Identity id)
            throws ObjectNotRegisteredException,
                   DeploymentException;
    ...
    }
}
```

- `addObject`
  The `addObject` operation adds a new object to the database. The proxy argument supplies the identity of the well-known object. If an object with the same identity has already been registered, the operation raises `ObjectExistsException`. Since this operation does not accept an argument supplying the object's type, the registry invokes `ice_id` on the given proxy to determine its most-derived type. The implication here is that the object must be available in order for the registry to obtain its type. If the object is not available, `addObject` raises `DeploymentException`.
- `updateObject`
  The `updateObject` operation supplies a new proxy for the well-known object whose identity is encapsulated by the proxy. If no object with the given identity is registered, the operation raises `ObjectNotRegisteredException`. The object's type is not modified by this operation.
- `addObjectWithType`
  The `addObjectWithType` operation behaves like `addObject`, except the object's type is specified explicitly and therefore the registry does not attempt to invoke `ice_id` on the given proxy (even if the type is an empty string).
- `removeObject`
  The `removeObject` operation removes the well-known object with the given identity from the database. If no object with the given identity is registered, the operation raises `ObjectNotRegisteredException`.

The following C++ example produces the same result as the descriptor we deployed earlier:

C++11 C++98

```
auto adapter = communicator->createObjectAdapter("EncoderAdapter");
auto ident = Ice::stringToIdentity("EncoderFactory");
auto f = make_shared<FactoryI>();
auto factory = adapter->add(f, ident);
std::shared_ptr<IceGrid::AdminPrx> admin = // ...
try
{
    admin->addObject(factory); // OOPS!
}
catch(const IceGrid::ObjectExistsException&)
{
    admin->updateObject(factory);
}
```

```
Ice::ObjectAdapterPtr adapter =
communicator->createObjectAdapter("EncoderAdapter");
Ice::Identity ident = Ice::stringToIdentity("EncoderFactory");
FactoryPtr f = new FactoryI;
Ice::ObjectPrx factory = adapter->add(f, ident);
IceGrid::AdminPrx admin = // ...
try
{
    admin->addObject(factory); // OOPS!
}
catch(const IceGrid::ObjectExistsException&)
{
    admin->updateObject(factory);
}
```

After obtaining a proxy for the `IceGrid::Admin` interface, the code invokes `addObject`. Notice that the code catches `ObjectExistsException` and calls `updateObject` instead when the object is already registered.

There is one subtle problem in this code: calling `addObject` causes the registry to invoke `ice_id` on our factory object, but we have not yet activated the object adapter. As a result, our program will hang indefinitely at the call to `addObject`. One solution is to activate the adapter prior to the invocation of `addObject`; another solution is to use `addObjectWithType` as shown below:

C++11 C++98

```
auto adapter = communicator->createObjectAdapter("EncoderAdapter");
auto ident = Ice::stringToIdentity("EncoderFactory");
auto f = std::make_shared<FactoryI>();
auto factory = adapter->add(f, ident);
std::shared_ptr<IceGrid::AdminPrx> admin = // ...
try
{
    admin->addObjectWithType(factory, factory->ice_id());
}
catch(const IceGrid::ObjectExistsException&)
{
    admin->updateObject(factory);
}
```

```
Ice::ObjectAdapterPtr adapter =
communicator->createObjectAdapter("EncoderAdapter");
Ice::Identity ident = Ice::stringToIdentity("EncoderFactory");
FactoryPtr f = new FactoryI;
Ice::ObjectPrx factory = adapter->add(f, ident);
IceGrid::AdminPrx admin = // ...
try
{
    admin->addObjectWithType(factory, factory->ice_id());
}
catch(const IceGrid::ObjectExistsException&)
{
    admin->updateObject(factory);
}
```

Adding Well-Known Objects with `icegridadmin`

The icegridadmin utility provides commands that are the functional equivalents of the Slice operations for managing well-known objects. We can use the utility to manually register the `EncoderFactory` object from our descriptors:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> object add "EncoderFactory@EncoderAdapter"
```

Use the `object list` command to verify that the object was registered successfully:

```
>>> object list
EncoderFactory
IceGrid/Query
IceGrid/Locator
IceGrid/Registry
IceGrid/InternalRegistry-Master
```

To specify the object's type explicitly, append it to the `object add` command:

```
>>> object add "EncoderFactory@EncoderAdapter"
"::Ripper::MP3EncoderFactory"
```

Finally, the object is removed from the registry like this:

```
>>> object remove "EncoderFactory"
```

Querying Well-Known Objects

The registry's database of well-known objects is not used solely for resolving indirect proxies. The database can also be queried interactively to find objects in a variety of ways. The `IceGrid::Query` interface supplies this functionality:

Slice

```
module IceGrid
{
    enum LoadSample
    {
        LoadSample1,
        LoadSample5,
        LoadSample15
    }

    interface Query
    {
        idempotent Object* findObjectById(Ice::Identity id);
        idempotent Object* findObjectByType(string type);

  idempotent Object* findObjectByTypeOnLeastLoadedNode(string type, Loa
dSample sample);

  idempotent Ice::ObjectProxySeq findAllObjectsByType(string type);
        idempotent Ice::ObjectProxySeq findAllReplicas(Object* proxy);
    }
}
```

- `findObjectById`
  The `findObjectById` operation returns the proxy associated with the given identity of a well-known object. It returns a null proxy if no match was found.
- `findObjectByType`
  The `findObjectByType` operation returns a proxy for an object registered with the given type. If more than one object has the same type, the registry selects one at random. The operation returns a null proxy if no match was found.
- `findObjectByTypeOnLeastLoadedNode`
  The `findObjectByTypeOnLeastLoadedNode` operation considers the system load when selecting one of the objects with the given type. If the registry is unable to determine which node hosts an object (for example, because the object was registered with a direct proxy and not an adapter ID), the object is considered to have a load value of `1` for the purposes of this operation. The sample argument determines the interval over which the loads are averaged (one, five, or fifteen minutes). The operation returns a null proxy if no match was found.
- `findAllObjectsByType`
  The `findAllObjectsByType` operation returns a sequence of proxies representing the well-known objects having the given type. The operation returns an empty sequence if no match was found.
- `findAllReplicas`
  Given an indirect proxy for a replicated object, the `findAllReplicas` operation returns a sequence of proxies representing the individual replicas. An application can use this operation when it is necessary to communicate directly with one or more replicas.

Be aware that the operations accepting a `type` parameter are not equivalent to invoking `ice_isA` on each object to determine whether it supports the given type, a technique that would not scale well for a large number of registered objects. Rather, the operations simply compare the given type to the object's registered type or, if the object was registered without a type, to the object's most-derived Slice type as determined by the registry.

> Starting with Ice 3.7, the find by type functions now only return proxies for well-known objects from servers which are enabled or proxies not registered through the deployment descriptors.

Using Well-Known Objects in the Ripper Application

Well-known objects are another IceGrid feature we can incorporate into our ripper application.

Adding Well-Known Objects to the Ripper Deployment

First we'll modify the descriptors to add two well-known objects:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp">
                    <object identity="EncoderFactory1"
type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp">
                    <object identity="EncoderFactory2"
type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

At first glance, the addition of the well-known objects does not appear to simplify our client very much. Rather than selecting which of the two adapters receives the next task, we now need to select one of the well-known objects.

Querying Ripper Objects with `findAllObjectsByType`

The `IceGrid::Query` interface provides a way to eliminate the client's dependency on object adapter identifiers and object identities. Since our factories are registered with the same type, we can search for all objects of that type:

C++11 C++98

```
auto proxy = communicator->stringToProxy("IceGrid/Query");
auto query = Ice::checkedCast<IceGrid::QueryPrx>(proxy);
string type = Ripper::MP3EncoderFactory::ice_staticId();
auto seq = query->findAllObjectsByType(type);
if(seq.empty())
{
    // no match
}
Ice::ObjectProxySeq::size_type index = ... // random number
auto factory =
Ice::checkedCast<Ripper::MP3EncoderFactoryPrx>(seq[index]);
auto encoder = factory->createEncoder();
```

```
Ice::ObjectPrx proxy = communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
string type = Ripper::MP3EncoderFactory::ice_staticId();
Ice::ObjectProxySeq seq = query->findAllObjectsByType(type);
if(seq.empty())
{
    // no match
}
Ice::ObjectProxySeq::size_type index = ... // random number
Ripper::MP3EncoderFactoryPrx factory =
Ripper::MP3EncoderFactoryPrx::checkedCast(seq[index]);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

This example invokes `findAllObjectsByType` and then randomly selects an element of the sequence.

Querying Ripper Objects with `findObjectByType`

We can simplify the client further using `findObjectByType` instead, which performs the randomization for us:

C++11 C++98

```
auto proxy = communicator->stringToProxy("IceGrid/Query");
auto query = Ice::checkedCast<Grid::QueryPrx>(proxy);
string type = Ripper::MP3EncoderFactory::ice_staticId();
auto obj = query->findObjectByType(type);
if(!obj)
{
    // no match
}
auto factory = Ice::checkedCast<Ripper::MP3EncoderFactoryPrx>(obj);
auto encoder = factory->createEncoder();
```

```
Ice::ObjectPrx proxy = communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
string type = Ripper::MP3EncoderFactory::ice_staticId();
Ice::ObjectPrx obj = query->findObjectByType(type);
if(!obj)
{
    // no match
}
Ripper::MP3EncoderFactoryPrx factory =
Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

Querying Ripper Objects with `findObjectByTypeOnLeastLoadedNode`

So far the use of `IceGrid::Query` has allowed us to simplify our client, but we have not gained any functionality. If we replace the call to `findObjectByType` with `findObjectByTypeOnLeastLoadedNode`, we can improve the client by distributing the encoding tasks more intelligently. The change to the client's code is trivial:

C++11 C++98

```
auto proxy = communicator->stringToProxy("IceGrid/Query");
auto query = Ice::checkedCast<IceGrid::QueryPrx>(proxy);
string type = Ripper::MP3EncoderFactory::ice_staticId();
auto obj = query->findObjectByTypeOnLeastLoadedNode(type,
IceGrid::LoadSample1);
if(!obj)
{
    // no match
}
auto factory = Ice::checkedCast<Ripper::MP3EncoderFactoryPrx>(obj);
auto encoder = factory->createEncoder();
```

```
Ice::ObjectPrx proxy = communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
string type = Ripper::MP3EncoderFactory::ice_staticId();
Ice::ObjectPrx obj = query->findObjectByTypeOnLeastLoadedNode(type,
IceGrid::LoadSample1);
if(!obj)
{
    // no match
}
Ripper::MP3EncoderFactoryPrx factory =
Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

Ripper Progress Review

Incorporating intelligent load distribution is a worthwhile enhancement and is a capability that would be time consuming to implement ourselves. However, our current design uses only well-known objects in order to make queries possible. We do not really need the encoder factory object on each compute server to be individually addressable as a well-known object, a fact that seems clear when we examine the identities we assigned to them: `EncoderFactory1`, `EncoderFactory2`, and so on. IceGrid's replication features give us the tools we need to improve our design.

See Also

- Terminology
- Type IDs
- Object Descriptor Element
- IceGrid Administrative Sessions
- icegridadmin Command Line Tool
- Object Adapter Replication

IceGrid Templates

IceGrid templates simplify the task of creating the descriptors for an application. A template is a parameterized descriptor that you can instantiate as often as necessary, and they are descriptors in their own right. Templates are components of an IceGrid application and therefore they are stored in the registry's database. As such, their use is not restricted to XML files; templates can also be created and instantiated interactively using the graphical administration tool.

You can define templates for server and service descriptors. The focus of this section is server templates; we discuss service descriptors and templates in the context of IceBox integration.

On this page:

Server Templates

You may recall from a previous example that the XML description of our sample application defined two nearly identical servers:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2" exe="/opt/ripper/bin/server"
activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

This example is an excellent candidate for a server template. Equivalent definitions that incorporate a template are shown below:

XML

```
<icegrid>
    <application name="Ripper">
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <server id="EncoderServer${index}"
exe="/opt/ripper/bin/server" activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
index="2"/>
        </node>
    </application>
</icegrid>
```

We have defined a server template named `EncoderServerTemplate`. Nested within the `server-template` element is a server descriptor that defines an encoder server. The only difference between this `server` element and our previous example is that it is now parameterized: the template parameter `index` is used to form unique identifiers for the server and its adapter. The symbol `${index}` is replaced with the value of the `index` parameter wherever it occurs.

The template is instantiated by a `server-instance` element, which may be used anywhere that a `server` element is used. The server instance descriptor identifies the template to be instantiated, and supplies a value for the `index` parameter.

Although we have not significantly reduced the length of our XML file, we have made it more readable. And more importantly, deploying this server on additional nodes has become much easier.

Template Parameters

Parameters enable you to customize each instance of a template as necessary. The example above defined the `index` parameter with a different value for each instance to ensure that identifiers are unique. A parameter may also declare a default value that is used in the template if no value is specified for it. In our sample application the `index` parameter is considered mandatory and therefore should not have a default value, but we can illustrate this feature in another way. For example, suppose that the path name of the server's executable may change on each node. We can supply a default value for this attribute and override it when necessary:

XML

```
<icegrid>
    <application name="Ripper">
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath" default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}" exe="${exepath}"
activation="on-demand">
                <adapter name="EncoderAdapter" endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate" index="2"
                exepath="/opt/ripper-test/bin/server"/>
        </node>
    </application>
</icegrid>
```

As you can see, the instance on `Node1` uses the default value for the new parameter `exepath`, but the instance on `Node2` defines a different location for the server's executable.

Understanding the semantics of descriptor variables and parameters will help you add flexibility to your own IceGrid applications.

Adding Properties to a Server Instance

As we saw in the preceding section, template parameters allow you to customize each instance of a server template, and template parameters with default values allow you to define commonly used configuration options. However, you might want to have additional configuration properties for a given instance without having to add a parameter. For example, to debug a server instance on a specific node, you might want to start the server with the `Ice.Trace.Network` property set; it would be inconvenient to have to add a parameter to the template just to set that property.

To cater for such scenarios, it is possible to specify additional properties for a server instance without modifying the template. You can define such properties in the `server-instance` element, for example:

XML

```
<icegrid>
    <application>
        ...
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
index="2">
                <properties>
                    <property name="Ice.Trace.Network" value="2"/>
                </properties>
            </server-instance>
        </node>
    </application>
</icegrid>
```

This sets the `Ice.Trace.Network` property for a specific server.

Default Templates

The IceGrid registry can be configured to supply any number of default template descriptors for use in your applications. The configuration property `IceGrid.Registry.DefaultTemplates` specifies the path name of an XML file containing template definitions. One such template file is provided in the Ice distribution as `config/templates.xml`, which contains helpful templates for deploying Ice services such as IcePatch2 and Glacier2.

The template file must use the structure shown below:

XML

```
<icegrid>
    <application name="DefaultTemplates">
        <server-template id="EncoderServerTemplate">
            ...
        </server-template>
    </application>
</icegrid>
```

The name you give to the application is not important, and you may only define server and service templates within it. After configuring the registry to use this file, your default templates become available to every application that imports them.

The descriptor for each application indicates whether the default templates should be imported. (By default they are not imported.) If the templates are imported, they are essentially copied into the application descriptor and treated no differently than templates defined by the application itself. As a result, changes to the file containing default templates have no effect on existing application descriptors. In XML, the attribute `import-default-templates` determines whether the default templates are imported, as shown in the following example:

XML

```
<icegrid>
    <application name="Ripper" import-default-templates="true">
        ...
    </application>
</icegrid>
```

Using Templates with `icegridadmin`

The IceGrid administration tools allow you to inspect templates and instantiate new servers dynamically. First, let us ask `icegridadmin` to describe the server template we created earlier:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> server template describe Ripper EncoderServerTemplate
```

This command generates the following output:

```
server template `EncoderServerTemplate'
{
    parameters = `index exepath'
    server `EncoderServer${index}'
    {
        exe = `${exepath}'
        activation = `on-demand'
        properties
        {
            EncoderAdapter.Endpoints = `tcp'
        }
        adapter `EncoderAdapter'
        {
            id = `EncoderAdapter${index}'
            replica group id =
            endpoints = `tcp'
            register process = `false'
            server lifetime = `true'
        }
    }
}
```

Notice that the server ID is a parameterized value; it cannot be evaluated until the template is instantiated with values for its parameters.

Next, we can use `icegridadmin` to create an instance of the encoder server template on a new node:

```
>>> server template instantiate Ripper Node3 EncoderServerTemplate
index=3
```

The command requires that we identify the application, node and template, as well as supply any parameters needed by the template. The new server instance is permanently added to the registry's database, but if we intend to keep this configuration it is a good idea to update the XML description of our application to reflect these changes and avoid potential synchronization issues.

See Also

- Server Descriptor Element
- Server-Template Descriptor Element
- Server-Instance Descriptor Element
- icegridadmin Command Line Tool

IceBox Integration with IceGrid

IceGrid makes it easy to configure an IceBox server with one or more services.

On this page:

Deploying an IceBox Server

An IceBox server shares many of the same characteristics as other servers, but its special requirements necessitate a new descriptor. Unlike other servers, an IceBox server generally hosts multiple independent services, each requiring its own communicator instance and configuration file.

As an example, the following application deploys an IceBox server containing one service:

XML

```
<icegrid>
    <application name="IceBoxDemo">
        <node name="Node">
            <icebox id="IceBoxServer" exe="/opt/Ice/bin/icebox"
activation="on-demand">
                <service name="ServiceA" entry="servicea:create">
                    <adapter name="${service}" endpoints="tcp"/>
                </service>
            </icebox>
        </node>
    </application>
</icegrid>
```

It looks very similar to a server descriptor. The most significant difference is the service descriptor, which is constructed much like a server in that you can declare its attributes such as object adapters and configuration properties. The order in which services are defined determines the order in which they are loaded by the IceBox server.

The value of the adapter's `name` attribute needs additional explanation. The symbol `service` is one of the names reserved by IceGrid. In the context of a service descriptor, `${service}` is replaced with the service's name, and so the object adapter is also named `ServiceA`.

Service Templates

If you are familiar with templates in general, an IceBox service template is readily understandable:

XML

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                <adapter name="${service}" endpoints="default"/>
                <property name="${service}.Identity"
value="${server}-${service}"/>
            </service>
        </service-template>
        <node name="Node1">
            <icebox id="IceBoxServer" endpoints="default"
                exe="/opt/Ice/bin/icebox" activation="on-demand">
                <service-instance template="ServiceTemplate"
name="Service1"/>
            </icebox>
        </node>
    </application>
</icegrid>
```

In this application, an IceBox server is deployed on a node and has one service instantiated from the service template. Of particular interest is the `property` descriptor, which uses another reserved name `server` to form the property value. When the template is instantiated by the service instance descriptor, the symbol `${server}` is replaced with the name of the enclosing server, so the property definition expands as follows:

```
Service1.Identity=IceBoxServer-Service1
```

As with server instances, you can specify additional properties for the service instance without modifying the template. These properties can be defined in the `service-instance` element, as shown below:

XML

```
<icegrid>
    <application name="IceBoxApp">
        ...
        <node name="Node1">
            <icebox id="IceBoxServer"endpoints="default"
                exe="/opt/Ice/bin/icebox" activation="on-demand">
                <service-instance template="ServiceTemplate"
name="Service1">
                    <properties>
                        <property name="Ice.Trace.Network" value="1"/>
                    </properties>
                </service-instance>
            </icebox>
        </node>
    </application>
</icegrid>
```

Advanced Service Templates

A more sophisticated use of templates involves instantiating a service template in a server template:

XML

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                <adapter name="${service}" endpoints="default"/>
                <property name="${name}.Identity"
value="${server}-${name}"/>
            </service>
        </service-template>
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <icebox id="${id}" endpoints="default"
                exe="/opt/Ice/bin/icebox" activation="on-demand">
                <service-instance template="ServiceTemplate"
name="Service1"/>
            </icebox>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate"
id="IceBoxServer"/>
        </node>
    </application>
</icegrid>
```

This application is equivalent to our first example of service templates. Now, however, the process of deploying an identical server on several nodes has become much simpler.

If you need the ability to customize the configuration of a particular service instance, your server instance can define a property set that applies only to the desired service:

XML

```
<icegrid>
    <application name="IceBoxApp">
        <node name="Node1">
            <server-instance template="ServerTemplate"
id="IceBoxServer">
                <properties service="Service1">
                    <property name="Ice.Trace.Network" value="1"/>
                </properties>
            </server-instance>
        </node>
    </application>
</icegrid>
```

As this example demonstrates, the `service` attribute of the property set denotes the name of the target service.

See Also

- IceBox
- IceBox Descriptor Element
- Service Descriptor Element
- Service-Template Descriptor Element
- Server-Template Descriptor Element
- Properties Descriptor Element
- Using Descriptor Variables and Parameters

Object Adapter Replication

As an implementation of an Ice location service, IceGrid supports object adapter replication. An application defines its replica groups and their participating object adapters using descriptors, and IceGrid generates the server configurations automatically.

On this page:

Deploying a Replica Group

The descriptor that defines a replica group can optionally declare well-known objects as well as configure the group to determine its behavior during locate requests. Consider this example:

XML

```
<icegrid>
    <application name="ReplicaApp">
        <replica-group id="ReplicatedAdapter">
            <object identity="TheObject" type="::Demo::ObjectType"/>
        </replica-group>
        <node name="Node">
            <server id="ReplicaServer" activation="on-demand"
exe="/opt/replica/bin/server">
                <adapter name="TheAdapter" endpoints="default"
replica-group="ReplicatedAdapter"/>
            </server>
        </node>
    </application>
</icegrid>
```

The adapter's descriptor declares itself to be a member of the replica group `ReplicatedAdapter`, which must have been previously created by a replica group descriptor.

The replica group `ReplicatedAdapter` declares a well-known object so that an indirect proxy of the form `TheObject` is equivalent to the indirect proxy `TheObject@ReplicatedAdapter`. Since this trivial example defines only one adapter in the replica group, the proxy `TheOb ject` is also equivalent to `TheObject@TheAdapter`.

Replica Group Membership

An object adapter participates in a replica group by specifying the group's ID in the adapter's `ReplicaGroupId` configuration property. Identifying the replica group in the IceGrid descriptor for an object adapter causes the node to include the equivalent `ReplicaGroupId` prop erty in the configuration file it generates for the server.

By default, the IceGrid registry requires the membership of a replica group to be statically defined. When you create a descriptor for an object adapter that identifies a replica group, the registry adds that adapter to the group's list of valid members. During an adapter's activation, when it describes its endpoints to the registry, an adapter that also claims membership in a replica group is validated against the registry's internal list.

In a properly configured IceGrid application, this activity occurs without incident, but there are situations in which validation can fail. For example, adapter activation fails if an adapter's ID is changed without notifying the registry, such as by manually modifying the server configuration file that was generated by a node.

It is also possible for activation to fail when the IceGrid registry is being used solely as a location service, in which case descriptors have not

been created and therefore the registry has no advance knowledge of the replica groups or their members. In this situation, adapter activation causes the server to receive `NotRegisteredException` unless the registry is configured to allow dynamic registration, which you can do by defining the following property:

```
IceGrid.Registry.DynamicRegistration=1
```

With this configuration, a replica group is created implicitly as soon as an adapter declares membership in it, and any adapter is allowed to participate.

The use of dynamic registration often leads to the accumulation of obsolete replica groups and adapters in the registry. The IceGrid administration tools allow you to inspect and clean up the registry's state.

Using Replica Groups in the Ripper Application

Replication is a perfect fit for the ripper application. The collection of encoder factory objects should be treated as a single logical object, and replication makes that possible.

Adding a Replica Group to the Ripper Deployment

Adding a replica group descriptor to our application is very straightforward:

XML

```
<icegrid>
    <application name="Ripper">
        <replica-group id="EncoderAdapters">
            <object identity="EncoderFactory"
type="::Ripper::MP3EncoderFactory"/>
        </replica-group>
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath" default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}" exe="${exepath}"
activation="on-demand">
                <adapter name="EncoderAdapter"
replica-group="EncoderAdapters"
                    endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
index="2"/>
        </node>
    </application>
</icegrid>
```

The new descriptor adds the replica group called `EncoderAdapters` and registers a well-known object with the identity `EncoderFactory`. The adapter descriptor in the server template has been changed to declare its membership in the replica group.

Using a Replica Group in the Ripper Client

In comparison to the examples that demonstrated querying for well-known objects, the new version of our client has become much simpler:

C++11 C++98

```
auto obj = communicator->stringToProxy("EncoderFactory");
auto factory = Ice::checkedCast<Ripper::MP3EncoderFactoryPrx>(obj);
auto encoder = factory->createEncoder();
```

```
Ice::ObjectPrx obj = communicator->stringToProxy("EncoderFactory");
Ripper::MP3EncoderFactoryPrx factory =
Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

The client no longer needs to use the `IceGrid::Query` interface, but simply creates a proxy for a well-known object and lets the Ice run time transparently interact with the location service. In response to a locate request for `EncoderFactory`, the registry returns a proxy containing the endpoints of both object adapters. The Ice run time in the client selects one of the endpoints at random, meaning we have now lost some functionality compared to the prior example in which system load was considered when selecting an endpoint. We will learn how to rectify this situation in our discussion of load balancing.

See Also

- Terminology
- Replica-Group Descriptor Element
- Object Descriptor Element
- Well-Known Objects
- Load Balancing

Load Balancing

Replication is an important IceGrid feature but, when combined with load balancing, replication becomes even more useful.

IceGrid nodes regularly report the system load of their hosts to the registry. The replica group's configuration determines whether the registry actually considers system load information while processing a locate request. Its configuration also specifies how many replicas to include in the registry's response.

IceGrid's load balancing capability assists the client in obtaining an initial set of endpoints for the purpose of establishing a connection. Once a client has established a connection, all subsequent requests on the proxy that initiated the connection are normally sent to the same server without further consultation with the registry. As a result, the registry's response to a locate request can only be viewed as a snapshot of the replicas at a particular moment. If system loads are important to the client, it must take steps to periodically contact the registry and update its endpoints.

On this page:

Replica Group Load Balancing

A replica group descriptor optionally contains a load balancing descriptor that determines how system loads are used in locate requests. The load balancing descriptor specifies the following information:

- Type
  Several load balancing types are supported.
- Sampling interval
  One of the load balancing types considers system load statistics, which are reported by each node at regular intervals. The replica group can specify a sampling interval of one, five, or fifteen minutes. Choosing a sampling interval requires balancing the need for up-to-date load information against the desire to minimize transient spikes. On Unix platforms, the node reports the system's load average for the selected interval, while on Windows the node reports the CPU utilization averaged over the interval.
- Number of replicas
  The replica group can instruct the registry to return the endpoints of one (the default) or more object adapters. If the specified number *N* is larger than one, the proxy returned in response to a locate request contains the endpoints of at most *N* object adapters. If *N* is 0, the proxy contains the endpoints of all the object adapters. The Ice run time in the client selects one of these endpoints at random when establishing a connection.

For example, the descriptor shown below uses adaptive load balancing to return the endpoints of the two least-loaded object adapters sampled with five-minute intervals:

XML

```
<replica-group id="ReplicatedAdapter">
    <load-balancing type="adaptive" load-sample="5" n-replicas="2"/>
</replica-group>
```

The type must be specified, but the remaining attributes are optional.

> IceGrid ignores the object adapters of a disabled server when executing a locate request, meaning the client that initiated the locate request will not receive the endpoints for any of these object adapters.

You can optionally use custom load balancing strategies by installing replica group filters.

Load Balancing Types

A replica group can select one of the following load balancing types:

- Random
  Random load balancing selects the requested number of object adapters at random. The registry does not consider system load for a replica group with this type.
- Adaptive
  Adaptive load balancing uses system load information to choose the least-loaded object adapters over the requested sampling interval. This is the only load balancing type that uses sampling intervals.
- Round Robin
  Round robin load balancing returns the least recently used object adapters. The registry does not consider system load for a replica group with this type. Note that the round-robin information is not shared between registry replicas; each replica maintains its own notion of the "least recently used" object adapters.
- Ordered
  Ordered load balancing selects the requested number of object adapters by priority. A priority can be set for each object adapter member of the replica group. If you define several object adapters with the same priority, IceGrid will order these object adapters according to their order of appearance in the descriptor.

Choosing the proper type of load balancing is highly dependent on the needs of client applications. Achieving the desired load balancing and fail-over behavior may also require the cooperation of your clients. To that end, it is very important that you understand how and when the Ice run time uses a locator to resolve indirect proxies.

Using Load Balancing in the Ripper Application

The only change we need to make to the ripper application is the addition of a load balancing descriptor:

XML

```
<icegrid>
    <application name="Ripper">
        <replica-group id="EncoderAdapters">
            <load-balancing type="adaptive"/>
            <object identity="EncoderFactory"
type="::Ripper::MP3EncoderFactory"/>
        </replica-group>
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath" default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}" exe="${exepath}"
activation="on-demand">
                <adapter name="EncoderAdapter"
replica-group="EncoderAdapters"
                         endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
index="2"/>
        </node>
    </application>
</icegrid>
```

Using adaptive load balancing, we have regained the functionality we forfeited when we introduced replica groups. Namely, we now select the object adapter on the least-loaded node, and no changes are necessary in the client.

Interacting with Object Replicas

In some applications you may have a need for interacting directly with the replicas of an object. You might be tempted to call `ice_getEndpoints` on the proxy of a replicated object in an effort to obtain the endpoints of all replicas, but that is not the correct solution because the proxy is indirect and therefore contains no endpoints. The proper approach is to query well-known objects using the `findAllReplicas` operation.

Custom Load Balancing Strategies

The IceGrid registry allows you to plug in custom load balancing implementations that the registry invokes to filter its query results. Two kinds of filters are supported:

- Replica group filter
  The registry invokes a replica group filter each time a client requests the endpoints of a replica group or object adapter, as well as for calls to `findAllReplicas`. The registry passes information about the query that the filter can use in its implementation,

including the list of object adapters participating in the replica group whose nodes are active at the time of the request. The object adapter list is initially ordered using the load balancing type configured for the replica group; the filter can modify this list however it chooses.

- Type filter
  The registry invokes a type filter for each query that a client issues to find a well-known object by type using the operations `findObjectByType`, `findAllObjectsByType`, and `findObjectByTypeOnLeastLoadedNode`. Included in the information passed to the filter is a list of proxies for the matching well-known objects; the filter implementation decides which of these proxies are returned to the client.

In the sections below we describe how to implement these filters.

Overview of Custom Load Balancing

Filters are installed into the IceGrid registry using the standard Ice plug-in facility. The registry is implemented in C++, using the Ice C++98 mapping, therefore the filters must be implemented in C++ with the C++98 mapping.

The Registry Plug-in Facade Object

During initialization, your plug-in will obtain a reference to a facade object with which it can register one or more filters. A filter typically retains a reference to this facade object because it offers a number of useful methods that the filter might need during its implementation. The `RegistryPluginFacade` class provides the following methods:

C++98

```
namespace IceGrid
{
    class RegistryPluginFacade : virtual public Ice::LocalObject
    {
    public:
        void addReplicaGroupFilter(const std::string& id, const
IceGrid::ReplicaGroupFilterPtr& filter);
        bool removeReplicaGroupFilter(const std::string& id, const
IceGrid::ReplicaGroupFilterPtr& filter);

        void addTypeFilter(const std::string& id, const
IceGrid::TypeFilterPtr& filter);
        bool removeTypeFilter(const std::string& id, const
IceGrid::TypeFilterPtr& filter);

        IceGrid::ApplicationInfo getApplicationInfo(const std::string&
name) const;

        IceGrid::ServerInfo getServerInfo(const std::string& serverId)
const;

        std::string getAdapterServer(const std::string& adapterId)
const;
        std::string getAdapterApplication(const std::string& adapterId)
const;
        std::string getAdapterNode(const std::string& adapterId) const;
        IceGrid::AdapterInfoSeq getAdapterInfo(const std::string&
adapterId) const;
        std::string getPropertyForAdapter(const std::string& adapterId,
const std::string& name) const;

        IceGrid::ObjectInfo getObjectInfo(const Ice::Identity& id)
const;

        IceGrid::NodeInfo getNodeInfo(const std::string& name) const;
        IceGrid::LoadInfo getNodeLoad(const std::string& name) const;
    }
    typedef ... RegistryPluginFacadePtr;

    RegistryPluginFacadePtr getRegistryPluginFacade();
}
```

There are methods for adding and removing replica group and type filters, along with a number of methods for obtaining information about the deployment. As you can see, a great deal of information is available to a filter implementation for use in making its decisions. The data structures returned by these methods correspond directly to their XML descriptors; you can also review the Slice definitions of the IceGrid data types for more information.

The methods are described below:

- `void addReplicaGroupFilter(const std::string& id, const IceGrid::ReplicaGroupFilterPtr& filter)`
  Adds a replica group filter with the given identifier. The identifier must match the `filter` attribute of a `replica-group` descriptor. The registry maintains a list of filters for each identifier. If you register more than one filter with the same identifier, the registry invokes each one in turn, in the order of registration. To add a replica group filter for dynamically registered replica groups, you should use the empty string for the identifier.
- `bool removeReplicaGroupFilter(const std::string& id, const IceGrid::ReplicaGroupFilterPtr& filter)`
  Removes an existing replica group filter that matches the given identifier and smart pointer. Returns true if a match was found, false otherwise.
- `void addTypeFilter(const std::string& id, const IceGrid::TypeFilterPtr& filter)`
  Adds a type filter with the given identifier. The registry maintains a list of filters for each identifier. If you register more than one filter with the same identifier, the registry invokes each one in turn, in the order of registration.
- `bool removeTypeFilter(const std::string& id, const IceGrid::TypeFilterPtr& filter)`
  Removes an existing type filter that matches the given identifier and smart pointer. Returns true if a match was found, false otherwise.
- `IceGrid::ApplicationInfo getApplicationInfo(const std::string& name) const`
  Returns the descriptor for the application with the given name. Raises `IceGrid::ApplicationNotExistException` if no match is found.
- `IceGrid::ServerInfo getServerInfo(const std::string& serverId) const`
  Returns the descriptor for the server with the given identifier. Raises `IceGrid::ServerNotExistException` if no match is found.
- `std::string getAdapterServer(const std::string& adapterId) const`
  Returns the identifier of the server hosting the object adapter with the given adapter identifier. Returns an empty string if no match is found.
- `std::string getAdapterApplication(const std::string& adapterId) const`
  Returns the name of the application containing the given object adapter identifier. Returns an empty string if no match is found.
- `std::string getAdapterNode(const std::string& adapterId) const`
  Returns the name of the node containing the given object adapter identifier. Returns an empty string if no match is found.
- `IceGrid::AdapterInfoSeq getAdapterInfo(const std::string& adapterId) const`
  If `adapterId` is a replica group identifier, this method returns a sequence of adapter descriptors for all of the replicas. Otherwise, this method returns a sequence containing one descriptor for the given object adapter. Raises `IceGrid::AdapterNotExistException` if no match is found.
- `std::string getPropertyForAdapter(const std::string& adapterId, const std::string& name) const`
  Obtains the value of a server configuration property with the key `name` for the server hosting the object adapter identified by `adapterId`. Returns an empty string if no match is found.
- `IceGrid::ObjectInfo getObjectInfo(const Ice::Identity& id) const`
  Returns information about an object with the given identity. Raises `IceGrid::ObjectNotRegisteredException` if no match is found.
- `IceGrid::NodeInfo getNodeInfo(const std::string& name) const`
  Returns the descriptor for the node with the given name. Raises `IceGrid::NodeNotExistException` if no match is found.
- `IceGrid::LoadInfo getNodeLoad(const std::string& name) const`
  Returns load information for the node with the given name. Raises `IceGrid::NodeNotExistException` if no match is found.

Methods that require access to deployment information raise `IceGrid::RegistryUnreachableException` if called before the registry has fully initialized itself.

Implementing a Registry Plug-in

The API for creating an Ice plug-in using C++ requires a factory function with external linkage, along with a class that implements the `Ice::Plugin` local Slice interface. We can use the code from the example in `demo/IceGrid/customLoadBalancing` to illustrate these points. First, the factory function instantiates and returns the plug-in:

C++98

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
    createRegistryPlugin(const Ice::CommunicatorPtr& communicator, const
string&, const Ice::StringSeq&)
    {
        return new RegistryPluginI(communicator);
    }
}
```

The plug-in class is straightforward:

C++98

```
class RegistryPluginI : public Ice::Plugin
{
public:
    RegistryPluginI(const Ice::CommunicatorPtr& communicator)
        : _communicator(communicator)
    {
    }

    virtual void initialize()
    {
        IceGrid::RegistryPluginFacadePtr facade =
IceGrid::getRegistryPluginFacade();
        if(facade)
        {
            facade->addReplicaGroupFilter("filterByCurrency", new
ReplicaGroupFilterI(facade));
        }
    }

    virtual void destroy()
    {
    }

private:
    const Ice::CommunicatorPtr _communicator;
};
```

The `initialize` method calls `getRegistryPluginFacade` to obtain a smart pointer for the registry's facade object. The plug-in uses this object to install a replica group filter.

We describe filter implementations in more detail below.

Installing a Registry Plug-in

Continuing with our example from `demo/IceGrid/customLoadBalancing`, we use the following property to install our plug-in in the IceGrid registry:

```
Ice.Plugin.RegistryPlugin=RegistryPlugin:createRegistryPlugin
```

The `Ice.Plugin` property must be defined in the registry's configuration file.

> Make sure to configure all of the replicas to load the same registry plug-ins, otherwise a client could get different behavior depending on which replica it's currently using.

Filter Implementation Techniques

A filter may require client-specific information in order to assemble its list of results. We recommend using request contexts for this purpose. Briefly, a request context is a dictionary of key/value string pairs that a client can configure and send along as "out of band" metadata accompanying a request. Ice provides several ways for a client to establish a request context:

- Implicit - provides a default request context for every request on all proxies
- Per-proxy - configures a default request context for every request on a particular proxy
- Explicit - specifies a request context at the time of invocation, overriding any default context

Our goal here is to transfer information from a client to a filter. Consequently, we don't recommend using implicit contexts because the context will add overhead to *every* invocation the client makes, not just the ones that involve the filter. To decide whether to use per-proxy or explicit contexts, we first must understand the circumstances in which each kind of filter receives a request context:

- Replica group filter

  Most invocations involving a replica group filter occur when the Ice run time in a client issues requests on a registry. This internal activity is triggered by the client's invocation on a proxy, but Ice doesn't use the client's proxy or the request context that the client may have configured for its proxy or passed explicitly to the invocation. Rather, the Ice run time uses the locator that the client configured for its proxy or communicator. As a result, the request contexts passed to a replica group filter are those configured for the *locator* proxy. (The only exception is when a client invokes `findAllReplicas` directly on the registry via its `Query` interface; refer to the discussion of type filters below for more details.)

  There are several ways you can configure a request context for the locator proxy. If a client configures its locator proxy statically using properties, the simplest solution is to add `Context` properties to the client's configuration, such as:

  ```
  Ice.Default.Locator=...
  Ice.Default.Locator.Context.someKey=someValue
  ```

  If you need to define the context at run time, you can obtain the locator proxy by calling `getDefaultLocator` on the communicator, create a new locator proxy with the desired context by calling `ice_context`, and then replace the locator proxy by calling `setDefaultLocator`.

  Both of these approaches are examples of per-proxy request contexts.

- Type filter

  Invocations involving type filters occur when a client invokes directly on the registry using its `Query` interface. To use per-proxy request contexts, configure the `Query` proxy as necessary. Explicit request contexts can also be used when querying the registry.

So far we've discussed how client-specific information can be passed to a filter, but what if the filter needs to obtain more information about the object adapters (in the case of a replica group filter) or objects (in the case of a type filter) in order to perform its duties? This is where the registry's facade object comes in handy, as with it the filter can retrieve information about the deployment. For example, a replica group filter can use server-specific properties as a form of metadata. The registry supplies the filter with a list of object adapter identifiers; each object adapter is hosted by a server, and the filter can look up property values for that server using the facade. The sample filter in `demo/IceGrid/customLoadBalancing` uses this technique, and we describe it in more detail below.

Implementing a Custom Replica Group Filter

A replica group filter must define a subclass of `ReplicaGroupFilter`:

C++98

```
namespace IceGrid
{
    class ReplicaGroupFilter : virtual public Ice::LocalObject
    {
    public:
        virtual Ice::StringSeq filter(const std::string& id,
                                      const Ice::StringSeq& adapters,
                                      const Ice::ConnectionPtr&
connection,
                                      const Ice::Context& context);
    };
}
```

The registry passes the following arguments to the `filter` method:

- `id`
  The replica group identifier involved in this request.
- `adapters`
  A sequence of object adapter identifiers denoting the object adapters participating in the replica group whose nodes are active at the time of the request. The object adapter list is initially ordered using the load balancing type configured for the replica group.
- `connection`
  The incoming connection from the client to the registry.
- `context`
  The request context that accompanied the request.

The implementation returns a sequence containing zero or more object adapter identifiers. The registry may truncate this list if the filter supplies more object adapters than the `n-replicas` value configured for the replica group's load balancing policy, but the registry will not change the ordering.

> The `filter` implementation must not block.

The C++ example in `demo/cpp98/IceGrid/customLoadBalancing` uses a replica group filter to select only those object adapters that support the currency requested by the client. The replica group's descriptor specifies the filter:

```
<replica-group id="ReplicatedPricingAdapter" filter="filterByCurrency">
  <load-balancing type="random"/>
  <object identity="pricing" type="::Demo::PricingEngine"/>
</replica-group>
```

Notice that the filter identifier `filterByCurrency` matches that used when the plug-in registered the filter.

In this example, the client uses a request context to indicate the desired currency. The context is configured on the locator proxy in the client's configuration file:

```
Ice.Default.Locator=...
Ice.Default.Locator.Context.currency=USD
```

Here we use the `Context` proxy property to statically assign a request context to the locator proxy, which means every invocation on the locator proxy includes the key/value pair `currency/USD`.

In addition to configuring the replica group filter, the deployment descriptor plays another important role here by defining server-specific properties that the filter uses in its implementation:

```
<server-template id="PricingServer">
  <parameter name="index"/>
  <parameter name="currencies"/>
  <server id="PricingServer-${index}" exe="./server"
activation="on-demand">
    <adapter name="Pricing" endpoints="tcp -h localhost"
replica-group="ReplicatedPricingAdapter"/>
    <property name="Identity" value="pricing"/>
    <property name="Currencies" value="${currencies}"/>
    ...
  </server>
</server-template>


<node name="node1">
  <server-instance template="PricingServer" index="1" currencies="EUR,
GBP, JPY"/>
  <server-instance template="PricingServer" index="2" currencies="USD,
GBP, AUD"/>
  <server-instance template="PricingServer" index="3" currencies="EUR,
USD, INR"/>
  <server-instance template="PricingServer" index="4" currencies="JPY,
GBP, AUD"/>
  <server-instance template="PricingServer" index="5" currencies="GBP,
AUD"/>
  <server-instance template="PricingServer" index="6" currencies="EUR,
USD"/>
</node>
```

As a convenience, the descriptor file uses a server template to define several servers, with each server supporting a specific set of currencies. The template adds to the property set of each server a property named `Currencies` representing the currencies that the server supports.

Finally, you can see how all of this ties together in the filter implementation:

C++98

```
Ice::StringSeq
ReplicaGroupFilterI::filter(const string& replicaGroupId,
                            const Ice::StringSeq& adapters,
                            const Ice::ConnectionPtr& connection,
                            const Ice::Context& ctx)
{
    Ice::Context::const_iterator p = ctx.find("currency");
    if(p == ctx.end())
    {
        return adapters;
    }

    string currency = p->second;

    Ice::StringSeq filteredAdapters;
    for(Ice::StringSeq::const_iterator p = adapters.begin(); p !=
adapters.end(); ++p)
    {
        if(_facade->getPropertyForAdapter(*p,
"Currencies").find(currency) != string::npos)
        {
            filteredAdapters.push_back(*p);
        }
    }
    return filteredAdapters;

}
```

The code first checks the request context for a value associated with the key `currency` and returns the adapter list unmodified if no entry is found. Next, the method builds a new list of adapters by iterating over the adapter list in its existing order and calling `getPropertyForAdapter` on the facade for each adapter. This method uses the registry's deployment information to find the server hosting the given adapter and then searches the server's configuration properties for one matching the given property name. If the `Currencies` property contains the client's specified currency, the adapter is added to the list that is eventually returned by the filter.

As this example demonstrates, request contexts are a convenient way to supply a filter with client-specific information, and server properties can serve as a simple database when a filter needs to tailor its results based on server attributes.

Implementing a Custom Type Filter

A replica group filter must define a subclass of `TypeFilter`:

C++98

```
namespace IceGrid
{
    class TypeFilter : virtual public Ice::LocalObject
    {
    public:
        virtual Ice::ObjectProxySeq filter(const std::string& type,
                                           const Ice::ObjectProxySeq&
proxies,
                                           const Ice::ConnectionPtr&
connection,
                                           const Ice::Context& context);
    };
}
```

The registry passes the following arguments to the `filter` method:

- `type`
  The type identifier involved in this request.
- `proxies`
  A sequence of proxies denoting the objects that matched the type.
- `connection`
  The incoming connection from the client to the registry.
- `context`
  The request context that accompanied the request.

The implementation returns a sequence containing zero or more proxies.

> The `filter` implementation must not block.

Refer to the previous section for more information on implementing a filter.

See Also

- Object Adapter Replication
- Connection Establishment
- Replica-Group Descriptor Element
- Load-Balancing Descriptor Element
- Well-Known Objects
- IceGrid Troubleshooting

Resource Allocation using IceGrid Sessions

IceGrid provides a resource allocation facility that coordinates access to the objects and servers of an IceGrid application. To allocate a resource for exclusive use, a client must first establish a session by authenticating itself with the IceGrid registry or a Glacier2 router, after which the client may reserve objects and servers that the application indicates are allocatable. The client should release the resource when it is no longer needed, otherwise IceGrid reclaims it when the client's session terminates or expires due to inactivity.

An allocatable server offers at least one allocatable object. The server is considered to be allocated when its first allocatable object is claimed, and is not released until all of its allocated objects are released. While the server is allocated by a client, no other clients can allocate its objects.

On this page:

Creating an IceGrid Session

A client must create an IceGrid session before it can allocate objects. If you have configured a Glacier2 router to use IceGrid's session managers, the client's router session satisfies this requirement.

In the absence of Glacier2, an IceGrid client invokes `createSession` or `createSessionFromSecureConnection` on IceGrid's `Registry` interface to create a session:

Slice

```
module IceGrid
{
    exception PermissionDeniedException
    {
        string reason;
    }

    interface Session extends Glacier2::Session
    {
        idempotent void keepAlive();
        ...
    }

    interface Registry
    {
        Session* createSession(string userId, string password)
            throws PermissionDeniedException;

        Session* createSessionFromSecureConnection()
            throws PermissionDeniedException;

        idempotent int getSessionTimeout();

        idempotent int getACMTimeout();

        ...
    }
}
```

The `createSession` operation expects a username and password and returns a session proxy if the client is allowed to create a session. By default, IceGrid does not allow the creation of sessions. You must define the registry property `IceGrid.Registry.PermissionsVerifier` with the proxy of a permissions verifier object to enable session creation with `createSession`.

The `createSessionFromSecureConnection` operation does not require a username and password because it uses the credentials supplied by an SSL connection to authenticate the client. As with `createSession`, you must enable session creation by configuring the proxy of a permissions verifier object so that clients can use `createSessionFromSecureConnection` to create a session. In this case, the property is `IceGrid.Registry.SSLPermissionsVerifier`.

To create a session, the client obtains the registry proxy by converting the well-known proxy string `"IceGrid/Registry"` to a proxy object with the communicator, downcasts the proxy to the `IceGrid::Registry` interface, and invokes one of the operations. The sample code below demonstrates how to do it in C++; the code will look very similar in other language mappings.

C++11 C++98

```
auto base = communicator->stringToProxy("IceGrid/Registry");
auto registry = Ice::checkedCast<IceGrid::RegistryPrx>(base);
string username = ...;
string password = ...;
shared_ptr<IceGrid::SessionPrx> session;
try
{
    session = registry->createSession(username, password);
}
catch(const IceGrid::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
```

```
Ice::ObjectPrx base = communicator->stringToProxy("IceGrid/Registry");
IceGrid::RegistryPrx registry =
IceGrid::RegistryPrx::checkedCast(base);
string username = ...;
string password = ...;
IceGrid::SessionPrx session;
try
{
    session = registry->createSession(username, password);
}
catch(const IceGrid::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
```

The identity of the registry object may change based on its configuration settings.

After creating the session, the client must keep it alive to prevent it from expiring. You have two options for keeping a session alive:

1. Call `Registry::getSessionTimeout` and periodically invoke `Session::keepAlive` every *timeout* (or less) seconds
2. Call `Registry::getACMTimeout` and configure ACM settings on the connection

We recommend using the second approach, which you can implement as follows:

C++11 C++98

```
int acmTimeout = registry->getACMTimeout();
if(acmTimeout > 0)
{
    auto conn = session->ice_getCachedConnection();
    conn->setACM(acmTimeout, Ice::nullopt,
Ice::ACMHeartbeat::HeartbeatAlways);
}
```

```
int acmTimeout = registry->getACMTimeout();
if(acmTimeout > 0)
{
    Ice::ConnectionPtr conn = session->ice_getCachedConnection();
    conn->setACM(acmTimeout, IceUtil::None, Ice::HeartbeatAlways);
}
```

Enabling heartbeats on the connection causes Ice to automatically send a heartbeat message at regular intervals determined by the given timeout value. The server ignores these messages, but they serve the purpose of keeping the session alive.

If a session times out, or if the client explicitly terminates the session by invoking its `destroy` operation, IceGrid automatically releases all objects allocated using that session.

Controlling Access to IceGrid Sessions

As described above, you must configure the IceGrid registry with the proxy of at least one permissions verifier object to enable session creation:

- `IceGrid.Registry.PermissionsVerifier`
  This property supplies the proxy of an object that implements the interface `Glacier2::PermissionsVerifier`. Defining this property allows clients to create sessions using `createSession`.
- `IceGrid.Registry.SSLPermissionsVerifier`
  This property supplies the proxy of an object that implements the interface `Glacier2::SSLPermissionsVerifier`. Defining this property allows clients to create sessions using `createSessionFromSecureConnection`.

IceGrid supplies built-in permissions verifier objects:

- A null permissions verifier for TCP/IP. This object accepts any username and password and should only be used in a secure environment where no access control is necessary. You select this verifier object by defining the following configuration property:

  ```
  IceGrid.Registry.PermissionsVerifier=<instance-name>/NullPermissio
  nsVerifier
  ```

  Note that you have to substitute the correct instance name for the object identity category.
- A null permissions verifier for SSL, analogous to the one for TCP/IP. You select this verifier object by defining the following configuration property:

  ```
  IceGrid.Registry.SSLPermissionsVerifier=<instance-name>/NullSSLPer
  missionsVerifier
  ```
- A file-based permissions verifier. This object uses an access control list in a file that contains username-password pairs. The format

of the password file is the same as the format of Glacier2 password files. You enable this verifier implementation by defining the configuration property `IceGrid.Registry.CryptPasswords` with the pathname of the password file. Note that this property is ignored if you specify the proxy of a permissions verifier object using `IceGrid.Registry.PermissionsVerifier`.

You can also implement your own permissions verifier object.

Allocating Objects with an IceGrid Session

A client allocates objects using the session proxy returned from `createSession` or `createSessionFromSecureConnection`. The proxy supports the `Session` interface shown below:

Slice

```
module IceGrid
{
    exception ObjectNotRegisteredException
    {
        Ice::Identity id;
    }

    exception AllocationException
    {
        string reason;
    }

    exception AllocationTimeoutException extends AllocationException
    {
    }

    interface Session extends Glacier2::Session
    {
        idempotent void keepAlive();

        Object* allocateObjectById(Ice::Identity id)
            throws ObjectNotRegisteredException, AllocationException;

        Object* allocateObjectByType(string type)
            throws AllocationException;

        void releaseObject(Ice::Identity id)
            throws ObjectNotRegisteredException, AllocationException;

        idempotent void setAllocationTimeout(int timeout);
    }
}
```

The client is responsible for keeping the session alive by periodically invoking `keepAlive`, as discussed earlier.

The `allocateObjectById` operation allocates and returns the proxy for the allocatable object with the given identity. If no allocatable object with the given identity is registered, the client receives `ObjectNotRegisteredException`. If the object cannot be allocated, the client receives `AllocationException`. An allocation attempt can fail for the following reasons:

- the object is already allocated by the session

- the object is allocated by another session and did not become available during the configured allocation timeout period
- the session was destroyed.

The `allocateObjectByType` operation allocates and returns a proxy for an allocatable object registered with the given type. If more than one allocatable object is registered with the given type, the registry selects one at random. The client receives `AllocationException` if no objects with the given type could be allocated. An allocation attempt can fail for the following reasons:

- no objects are registered with the given type
- all objects with the given type are already allocated (either by this session or other sessions) and none became available during the configured allocation timeout period
- the session was destroyed.

The `releaseObject` operation releases an object allocated by the session. The client receives `ObjectNotRegisteredException` if no allocatable object is registered with the given identity and `AllocationException` if the object is not allocated by the session. Upon session destruction, IceGrid automatically releases all allocated objects.

The `setAllocationTimeout` operation configures the timeout used by the allocation operations. If no allocatable objects are available when the client invokes `allocateObjectById` or `allocateObjectByType`, IceGrid waits for the specified timeout period for an allocatable object to become available. If the timeout expires, the client receives `AllocationTimeoutException`.

Allocating Servers with an IceGrid Session

A client does not need to explicitly allocate a server. If a server is allocatable, IceGrid implicitly allocates it to the first client that claims one of the server's allocatable objects. Likewise, IceGrid releases the server when all of its allocatable objects are released.

Server allocation is useful in two situations:

- Only allocatable servers can use the `session` activation mode, in which the server is activated on demand when allocated by a client and deactivated upon release.
- An allocatable server can be secured with IceSSL or Glacier2 so that its objects can only be invoked by the client that allocated it.

Security Considerations for Allocated Resources

IceGrid's resource allocation facility allows clients to coordinate access to objects and servers but does not place any restrictions on client invocations to allocated objects; any client that has a proxy for an allocated object could conceivably invoke an operation on it. IceGrid assumes that clients are cooperating with each other and respecting allocation semantics.

To prevent unauthorized clients from invoking operations on an allocated object or server, you can use IceSSL or Glacier2:

- Using IceSSL, you can secure access to a server or a particular object adapter with the properties `IceSSL.TrustOnly.Server` or `IceSSL.TrustOnly.Server.AdapterName`. For example, if you configure a server with the session activation mode, you can set one of the `IceSSL.TrustOnly` properties to the `${session.id}` variable, which is substituted with the session ID when the server is activated for the session. If the IceGrid session was created from a secure connection, the session ID will be the distinguished name associated with the secure connection, which effectively restricts access to the server or one of its adapters to the client that established the session with IceGrid.
- With Glacier2, you can secure access to an allocated object or the object adapters of an allocated server with the Glacier2 filtering mechanism. By default, IceGrid sessions created with a Glacier2 router are automatically given access to allocated objects, allocatable objects, certain well-known objects, and the object adapters of allocated servers.

Deploying Allocatable Resources

Allocatable objects are registered using a descriptor that is similar to well-known object descriptors. Allocatable objects cannot be replicated and therefore can only be specified within an object adapter descriptor.

Servers can be specified as allocatable by setting the server descriptor's `allocatable` attribute.

As an example, the following application defines an allocatable server and an allocatable object:

XML

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer"
                exe="/opt/ripper/bin/server"
                activation="on-demand"
                allocatable="true">
                <adapter name="EncoderAdapter" id="EncoderAdapter" endp
oints="tcp">
                    <allocatable identity="EncoderFactory" type="::Ripp
er::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

Using Resource Allocation in the Ripper Application

We can use the allocation facility in our MP3 encoder factory to coordinate access to the MP3 encoder factories. First we need to modify the descriptors to define an allocatable object:

XML

```
<icegrid>
    <application name="Ripper">
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <server id="EncoderServer${index}"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"  endpoints="tcp">
                    <allocatable identity="EncoderFactory${index}"
                        type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate" index="1"
/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate" index="2"
/>
        </node>
    </application>
</icegrid>
```

Next, the client needs to create a session and allocate a factory:

C++11 C++98

```
auto obj = session->allocateObjectByType(Ripper::MP3EncoderFactory::ice
_staticId());
try
{
    auto encoder = factory->createEncoder();
    // Use the encoder to encode a file ...
}
catch(const Ice::LocalException& ex)
{
    // There was a problem with the encoding, we catch the
    // exception to make sure we release the factory.
}
session->releaseObject(obj->ice_getIdentity());
```

```
Ice::ObjectPrx obj = session->allocateObjectByType(Ripper::MP3EncoderFa
ctory::ice_staticId());
try
{
    Ripper::MP3EncoderPrx encoder = factory->createEncoder();
    // Use the encoder to encode a file ...
}
catch(const Ice::LocalException& ex)
{
    // There was a problem with the encoding, we catch the
    // exception to make sure we release the factory.
}
session->releaseObject(obj->ice_getIdentity());
```

It is important to release an allocated object when it is no longer needed so that other clients may use it. If you forget to release an object, it remains allocated until the session is destroyed.

See Also

- Getting Started with Glacier2
- IceSSL
- Well-Known Registry Objects
- Securing a Glacier2 Router
- Object Descriptor Element
- Allocatable Descriptor Element
- IceGrid.*
- IceSSL.*

Registry Replication

The failure of an IceGrid registry or registry host can have serious consequences. A client can continue to use an existing connection to a server without interruption, but any activity that requires interaction with the registry is vulnerable to a single point of failure. As a result, the IceGrid registry supports replication using a master-slave configuration to provide high availability for applications that require it.

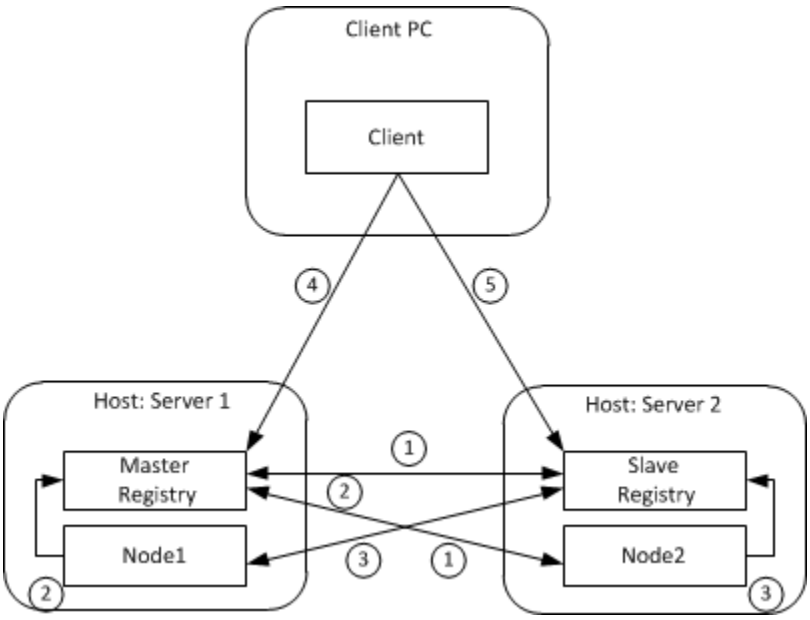
On this page:

Registry Replication Architecture

In IceGrid's registry replication architecture, there is one master replica and any number of slave replicas. The master synchronizes its deployment information with the slaves so that any replica is capable of responding to locate requests, managing nodes, and starting servers on demand. Should the master registry or its host fail, properly configured clients transparently fail over to one of the slaves.

Each replica has a unique name. The name `Master` is reserved for the master replica, while replicas can use any name that can legally appear in an object identity.

The figure below illustrates the underlying concepts of registry replication:



Overview of registry replication.

1. The slave replica contacts the master replica at startup and synchronizes its databases. Any subsequent modifications to the deployed applications are made via the master replica, which distributes them to all active slaves.
2. The nodes contact the master replica at startup to notify it about their availability.
3. The master replica provides a list of slave replicas to the nodes so that the nodes can also notify the slaves.
4. The client's configuration determines which replica it contacts initially. In this example, it contacts the master replica.
5. In the case of a failure, the client automatically fails over to the slave. If the master registry's host has failed, then `Node1` and any

servers that were active on this host also become unavailable. The use of object adapter replication allows the client to transparently reestablish communication with a server on `Node2`.

Capabilities of a Registry Replica

A master registry replica has a number of responsibilities, only some of which are supported by slaves. The master replica knows all of its slaves, but the slaves are not in contact with one another. If the master replica fails, the slaves can perform several vital functions that should keep most applications running without interruption. Eventually, however, a new master replica must be started to restore full registry functionality. For a slave replica to become the master, the slave must be restarted.

Locate Requests

One of the most important functions of a registry replica is responding to locate requests from clients, and every replica has the capability to service these requests. Slaves synchronize their databases with the master so that they have all of the information necessary to transform object identities, object adapter identifiers, and replica group identifiers into an appropriate set of endpoints.

Server Activation

Nodes establish sessions with each active registry replica so that any of the replicas are capable of activating a server on behalf of a client.

Queries

Replicating the registry also replicates the object that supports the `IceGrid::Query` interface used to query well-known objects. A client that resolves the `IceGrid/Query` object identity receives the endpoints of all active replicas, any of which can execute the client's requests.

Allocation

A client that needs to allocate a resource must establish a session with the master replica.

Administration

The state of an IceGrid registry is accessible via the `IceGrid::Admin` interface or (more commonly) using an administrative tool that encapsulates this interface. Modifications to the registry's state, such as deploying or updating an application, can only be done using the master replica. Administrative access to slave replicas is allowed but restricted to read-only operations. The administrative utilities provide mechanisms for you to select a particular replica to contact.

For programmatic access to a replica's administrative interface, the `IceGrid/Registry` identity corresponds to the master replica and the identity `IceGrid/Registry-`*`name`* corresponds to the slave with the given name.

Glacier2 Support

The registry implements the session manager interfaces required for integration with a Glacier2 router. The master replica supports the object identities `IceGrid/SessionManager` and `IceGrid/AdminSessionManager`. The slave replicas offer support for read-only administrative sessions using the object identity `IceGrid/AdminSessionManager-`*`name`*.

Configuring Registry Replication

Incorporating registry replication into an application is primarily accomplished by modifying your IceGrid configuration settings.

Replicas

Each replica must specify a unique name in its configuration property `IceGrid.Registry.ReplicaName`. The default value of this property is `Master`, therefore the master replica can omit this property if desired.

At startup, a slave replica attempts to register itself with its master in order to synchronize its databases and obtain the list of active nodes. The slave uses the proxy supplied by the `Ice.Default.Locator` property to find the master. At a minimum, this proxy should contain the

endpoint of a replica that is connected to the master.

For better reliability if a failure occurs, we recommend that you also include the endpoints of all slave replicas in the `Ice.Default.Locator` property. There is no harm in adding the slave's own endpoints to the proxy in `Ice.Default.Locator`; in fact, it makes configuration simpler because all of the slaves can share the same property definition. Although slaves do not communicate with each other, it is possible for one of the slaves to be promoted to the master, therefore supplying the endpoints of all slaves minimizes the chance of a communication failure.

Shown below is an example of the configuration properties for a master replica:

```
IceGrid.InstanceName=DemoIceGrid
IceGrid.Registry.Client.Endpoints=default -p 12000
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.LMDB.Path=db/master
...
```

You can configure a slave replica to use this master with the following settings:

```
Ice.Default.Locator=DemoIceGrid/Locator:default -p 12000
IceGrid.Registry.Client.Endpoints=default -p 12001
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.LMDB.Path=db/replica1
IceGrid.Registry.ReplicaName=Replica1
...
```

> Configuring IceLocatorDiscovery in the replicas allows them to discover the master at run time without the need to define `Ice.Default.Locator`.

Clients

The endpoints contained in the `Ice.Default.Locator` property determine which registry replicas the client can use when issuing locate requests. If high availability is important, this property should include the endpoints of at least two (and preferably all) replicas. Not only does this increase the reliability of the client, it also distributes the work load of responding to locate requests among all of the replicas.

Continuing the example from the previous section, you can configure a client with the `Ice.Default.Locator` property as shown below:

```
Ice.Default.Locator=DemoIceGrid/Locator:default -p 12000:default -p 120
01
```

> Configuring IceLocatorDiscovery in a client allows it to discover the replicas at run time without the need to define `Ice.Default.Locator`.

Nodes

As with slave replicas and clients, an IceGrid node should be configured with an `Ice.Default.Locator` property that contains the endpoint of at least one replica and preferably all the replicas. A node needs to notify each of the registry replicas about its presence, thereby enabling the replicas to activate its servers and obtain the endpoints of its object adapters. Only the master replica knows the list of active replica slaves so it's important that the node connects to the master on startup to retrieve the list of all the active replicas. If the master

is down when the node starts, the node will try to obtain the list of the registry replicas from the replicas specified in its `Ice.Default.Locator` proxy.

The following properties demonstrate how to configure a node with a replicated registry:

```
Ice.Default.Locator=DemoIceGrid/Locator:default -p 12000:default -p 120
01
IceGrid.Node.Name=node1
IceGrid.Node.Endpoints=default
IceGrid.Node.Data=db/node1
```

Configuring IceLocatorDiscovery in a node allows it to discover the replicas at run time without the need to define `Ice.Default.Locator`.

Diagnostics

You can use several configuration properties to enable trace messages that may help in diagnosing registry replication issues:

- `IceGrid.Registry.Trace.Replica`
  Displays information about the sessions established between master and slave replicas.
- `IceGrid.Registry.Trace.Node`
  `IceGrid.Node.Trace.Replica`
  Displays information about the sessions established between replicas and nodes.

Using Registry Replication with External Load Balancing

As explained earlier, we recommend including the endpoints of all replicas in the `Ice.Default.Locator` property. However, doing so might not always be convenient in large deployments, as it can require modifying many configuration files whenever you add or remove a registry replica. There are two ways to simplify your configuration:

- **Use a DNS name bound to multiple address (A) records**
  Configure the DNS name to point to the IP addresses of each of the registry replicas. You can then define the `Ice.Default.Locator` property with a single endpoint that embeds the DNS name. The Ice run time in the client randomly picks one of these IP addresses when it resolves the DNS name. All replicas must use the same port.
- **Use a TCP load balancer**
  Configure the load balancer to redirect traffic to the registry replicas and define the `Ice.Default.Locator` property with an endpoint that embeds the IP address and port of the TCP load balancer. The Ice run time in the client connects to the load balancer and the load balancer forwards the traffic to an active registry replica.

For maximum reliability, the DNS server or TCP load balancer should also be replicated.

When using such a configuration for the `Ice.Default.Locator` property of registry slaves, special care needs to be taken when initially starting the IceGrid registry replicas. Since the `Ice.Default.Locator` property no longer includes the endpoint of the master replica, a slave replica won't be able to contact the master if it's started when no master is running. You must first start the master and then the slaves to prevent this from happening. Once a replica slave connects successfully to the master, it saves the endpoint of the master and the other slaves in its database, so this is really only an issue when starting a slave with an empty database.

See Also

- Well-Known Objects
- icegridadmin Command Line Tool
- Glacier2 Integration with IceGrid
- IceGrid.*

Application Distribution

In the section so far, "deployment" has meant the creation of descriptors in the registry. A broader definition involves a number of other tasks:

- Writing IceGrid configuration files and preparing data directories on each computer
- Installing the IceGrid binaries and dependent libraries on each computer
- Starting the registry and/or node on each computer, and possibly configuring the systems to launch them automatically
- Distributing your server executables, dependent libraries and supporting files to the appropriate nodes.

Large and small scale applications often use tools to manage their deployments. The functionally of these tools range from doing basic tasks, like syncing files, to orchestrating complex tasks such as installing packages, configuring services, etc. We will assume that you have already done the first three tasks.

Topics

Application Distribution with Ansible

On this page:

Using Ansible to Distribute Applications

Ansible is a simple and easy to use automation tool which uses SSH for authentication and communication between the control machine and hosts, allowing you to distribute applications to many hosts with very little setup. Tasks to be performed are designed as playbooks – Ansible 's configuration, deployment, and orchestration language. Let's look at how we can use Ansible to securly distribute applications to IceGrid.

IceGrid Configuration

In this example we assume that you have already configured and deployed your servers using IceGrid. Consider the following configuration:

IceGrid Configuration

```
<icegrid>
    <application name="MyDemoApp">
        <node name="Node1">
            <server id="ServerA1" exe="/path/to/application/serverA"
...>
            </server>
            <server id="ServerB1" exe="/path/to/application/serverB"
...>
            </server>
        </node>
        <node name="Node2">
            <server id="ServerA2" exe="/path/to/application/serverA"
...>
            </server>
        </node>
        ...
    </application>
</icegrid>
```

This deployment contains three servers:

- `ServerA1`: located at `/path/to/application/serverA`, running on `Node1`
- `ServerA2`: located at `/path/to/application/serverA`, running on `Node2`
- `ServerB1`: located at `/path/to/application/serverB`, running on `Node1`

We will also assume you're running one IceGrid registry master instance, and two slave instances.

Ansible Configuration

We distribute our server applications in three steps using an Ansible playbook:

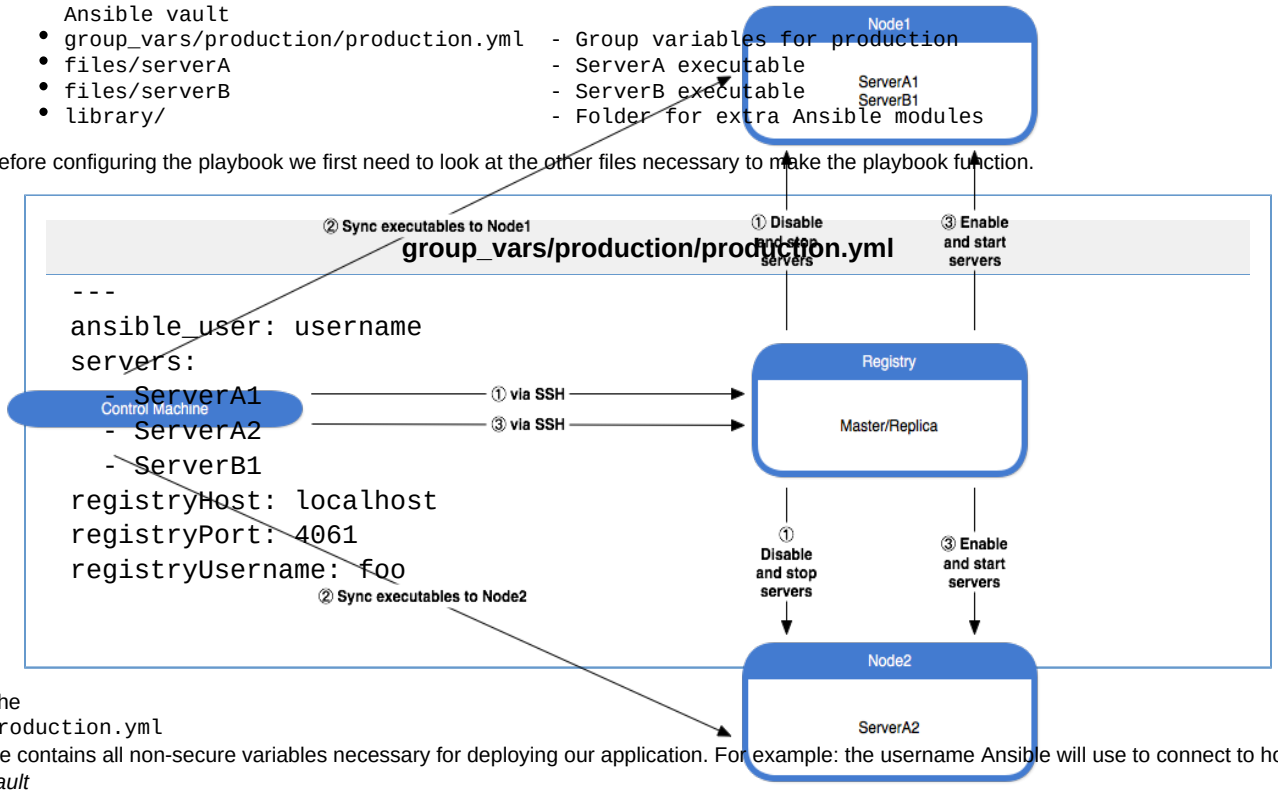
1. Disable and stop both servers using the IceGrid registry. We want to ensure the servers are disabled so that IceGrid does not automa
2. Update the server executables on each node.
3. Re-enable and start the servers using the IceGrid registry.

To do this we will need a directory with the following layout and files on your local system (where you will run Ansible playbooks).

- `deploy-server.yml` - Ansible playbook
- `inventories/production` - Ansible inventory file
- `group_vars/production/secure.yml` - Encrypted

  Ansible vault
- `group_vars/production/production.yml` - Group variables for production
- `files/serverA` - ServerA executable
- `files/serverB` - ServerB executable
- `library/` - Folder for extra Ansible modules

Before configuring the playbook we first need to look at the other files necessary to make the playbook function.



The `production.yml` file contains all non-secure variables necessary for deploying our application. For example: the username Ansible will use to connect to hosts

vault

. Vaults are stored on disk encrypted and are only decrypted and loaded into memory while a playbook is running. To create a vault run the f

```
ansible-vault create group_vars/production/secure.yml
```

After answering all of the prompted questions you can enter data into the newly created vault (your default editor will be automatically opened

group_vars/production/secure.yml

```
---
regiistryPassword: supersecretpassphrase
```

Be sure to save the file when you are finished. You can edit it later by running:

```
ansible-vault edit group_vars/production/secure.yml
```

Next is the ansible inventory file. This is an INI formatted file which corresponds to groups of hosts on which Ansible will run tasks. Ansible inventory files are used to map these groups to actual IP addresses or DNS entries. A typical deployment will contain at least two inventory files: one for production and one for testing/staging. The following is an example of a *production* inventory file:

inventories/production

```
[registry-master]
10.0.0.10

[registry-replicas]
10.0.0.11
10.0.0.12

[server-a]
10.0.0.20
10.0.0.21

[server-b]
10.0.0.20

[registries:children]
registry-master
registry-replicas

[nodes:children]
server1
server2

[production:children]
registries
nodes
```

Our production inventory contains a master IceGrid registry (`registry-master`) with two replicas (`registry-replicas`), as well as hosts that correspond to the servers which run the `serverA` application (`server-a`) and the `serverB` application (`server-b`). You can also establish host specific settings (such as login username) in this inventory file.

We are now ready to write the playbook. The following playbook distributes a server executable to each IceGrid node using the logic described above.

icegrid_servers module
The `icegrid_servers` module from ZeroC's ice-ansible repository is required by this playbook. It can be installed by copying `icegrid_servers.yml` into the `library` folder.

deploy-server.yml

```
---
#
# Disable and stop servers, synchronize server executables, and then
enable and start servers.
#
- hosts: registries
  tasks:
 - name: Stop and disable servers
   icegrid_servers:
   servers: "{{ servers }}"
     username: "{{ registryUsername }}"
        password: "{{ registryPassword }}"
        enabled: no
        state: stopped
   run_once: true

- hosts: server-a
  tasks:
    - name: Synchronize serverA application
      synchronize: src=serverA dest=/path/to/application/serverA

- hosts: server-b
  tasks:
    - name: Synchronize serverB application
      synchronize: src=serverB dest=/path/to/application/serverB

- hosts: registries
  tasks:
    - name: Enable and start servers
    icegrid_servers:
   servers: "{{ servers }}"
     username: "{{ registryUsername }}"
        password: "{{ registryPassword }}"
        enable: yes
        state: started
      run_once: true
```

The first and last tasks are performed on the first host in the `registries` (see run_once documentation) group of the inventory, while the second and third tasks are performed on the `server-1` and `server-2` groups, respectfully.

Running the Playbook

To run the `distribute-server.yml` playbook on the `production` inventory, execute the following command:

```
ansible-playbook -i inventories/production deploy-server.yml
--ask-vault-pass
```

If your master IceGrid registry is unavailable you can filter it out of play by running:

```
ansible-playbook -i inventories/production deploy-server.yml
--ask-vault-pass --limit "!registry-master"
```

The registry tasks will now run on the next available registry.

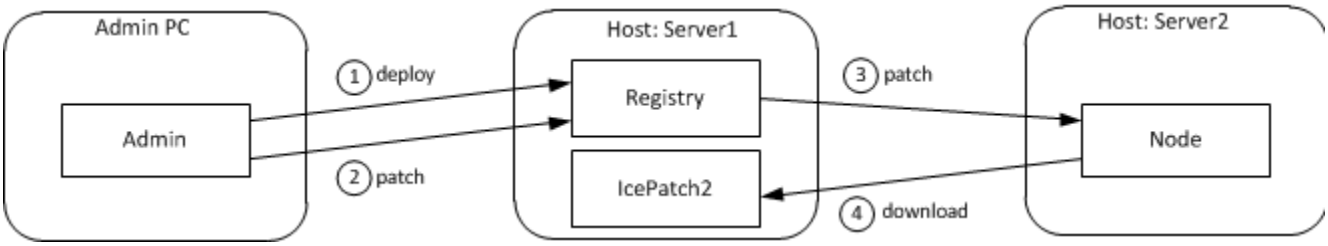
Application Distribution with IcePatch2

On this page:

Deprecation Notice
IcePatch2 has been deprecated and will be removed in a future version of Ice.

Using IcePatch2 to Distribute Applications

Using an IcePatch2 server, you can configure the nodes to download servers automatically and patch them at any time. The illustration below shows the interactions of the components:



Overview of application distribution.

As you can see, deploying an IceGrid application has greater significance when IcePatch2 is also involved. After deployment, the administrative tool initiates a patch, causing the registry to notify all active nodes that are configured for application distribution to begin the patching process. Since each IceGrid node is an IcePatch2 client, the node performs the patch just like any IcePatch2 client: it downloads everything if no local copy of the distribution exists, otherwise it does an incremental patch in which it downloads only new files and those whose signatures have changed.

The benefits of this feature are clear:

- The distribution files are maintained in a central location
- Updating a distribution on all of the nodes is a simple matter of preparing the master distribution and letting IceGrid do the rest
- Manually transferring executables and supporting files to each computer is avoided, along with the mistakes that manual intervention sometimes introduces.

Deploying an IcePatch2 Server

If you plan to use IceGrid's distribution capabilities, we generally recommend deploying an IcePatch2 server along with your application. Doing so gives you the same benefits as any other IceGrid server, including on-demand activation and remote administration. We will only use one server in our sample application, but you might consider replicating a number of IcePatch2 servers in order to balance the patching load for large distributions.

Patching Considerations

Deploying an IcePatch2 server with your application presents a chicken-and-egg dilemma: how do the nodes download their distributions if the IcePatch2 server is included in the deployment? To answer this question, we need to learn more about IceGrid's behavior.

Deploying and patching are treated as two separate steps: first you deploy the application, then you initiate the patching process. The `icegridadmin` utility combines these steps into one command (`application add`), but also provides an option to disable the patching step if so desired.

Let's consider the state of the application after deployment but before patching: we have described the servers that run on each node, including file system-dependent attributes such as the pathnames of their executables and default working directories. If these pathnames refer to directories in the distribution, and the distribution has not yet been downloaded to that node, then clearly we cannot attempt to use those servers until patching has completed. Similarly, we cannot deploy an IcePatch2 server whose executable resides in the distribution to be downloaded.

> We are ignoring the case where a temporary IcePatch2 server is used to bootstrap other IcePatch2 servers.

For these reasons, we assume that the IcePatch2 server and supporting libraries are distributed by the system administrator along with the IceGrid registry and nodes to the appropriate computers. The server should be configured for on-demand activation so that its node starts it automatically when patching begins. If the server is configured for manual activation, you must start it prior to patching.

IcePatch2 Server Template

The Ice distribution includes an IcePatch2 server template that simplifies the inclusion of IcePatch2 in your application. The relevant portion from the file `config/templates.xml` is shown below:

XML

```
<server-template id="IcePatch2">
    <parameter name="instance-name" default="${application}.IcePatch2"/>
    <parameter name="endpoints" default="default"/>
    <parameter name="directory"/>

    <server id="${instance-name}" exe="icepatch2server"
        application-distrib="false" activation="on-demand">
        <adapter name="IcePatch2" endpoints="${endpoints}">
            <object identity="${instance-name}/server"
type="::IcePatch2::FileServer"/>
        </adapter>
        <adapter name="IcePatch2.Admin" id=""
endpoints="tcp -h 127.0.0.1"/>
        <property name="IcePatch2.InstanceName"
value="${instance-name}"/>
        <property name="IcePatch2.Directory" value="${directory}"/>
    </server>
</server-template>
```

Notice that the server's pathname is `icepatch2server`, meaning the program must be present in the node's executable search path. The only mandatory parameter is `directory`, which specifies the server's data directory and becomes the value of the `IcePatch2.Directory` property. The value of the `instance-name` parameter is used as the server's identifier when the template is instantiated; its default value includes the name of the application in which the template is used. This identifier also affects the identities of the two well-known objects declared by the server.

Consider the following sample application:

XML

```
<icegrid>
    <application name="PatchDemo">
        <node name="Node">
            <server-instance template="IcePatch2"
directory="/opt/icepatch2/data"/>
            ...
        </node>
    </application>
</icegrid>
```

Instantiating the `IcePatch2` template creates a server identified as `PatchDemo.IcePatch2` (as determined by the default value for the `instance-name` parameter). The well-known objects use this value as the category in their identities, such as `PatchDemo.IcePatch2/server`.

In order to refer to the `IcePatch2` template in your application, you must have already configured the registry to use the `config/templates.xml` file as your default templates, or copied the template into the XML file describing your application.

Adding Distribution to a Deployment

A distribution descriptor provides the details that a node requires in order to download the necessary files. Specifically, the descriptor supplies the proxy of the IcePatch2 server and the names of the subdirectories comprising the distribution, all of which are optional. If the descriptor does not define the proxy, the following default value is used instead:

```
${application}.IcePatch2/server
```

You may recall that this value matches the default identity configured by the IcePatch2 server template described above. Also notice that this is an indirect proxy, implying that the IcePatch2 server was deployed with the application and can be started on-demand if necessary.

If the descriptor does not select any subdirectories, the node downloads the entire contents of the IcePatch2 data directory.

In XML, a descriptor having the default behavior as described above can be written as shown below:

XML

```
<distrib/>
```

To specify a proxy, use the `icepatch` attribute:

XML

```
<distrib icepatch="PatchDemo.IcePatch2/server"/>
```

Finally, select subdirectories using a nested element:

XML

```
<distrib>
    <directory>dir1</directory>
    <directory>dir2/subdir</directory>
</distrib>
```

By including only certain subdirectories in a distribution, you are minimizing the time and effort required to download and patch each node. For example, each node in a heterogeneous network might download a platform-specific subdirectory and another subdirectory containing files common to all platforms.

Distributing Applications and Servers

A distribution descriptor can be used in two contexts: within an application, and within a server. When the descriptor appears at the application level, it means every node in the application downloads that distribution. This is useful for distributing files required by all of the nodes on which servers are deployed, especially in a grid of homogeneous computers where it would be tedious to repeat the same distribution information in each server descriptor. Here is a simple XML example:

XML

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        ...
    </application>
</icegrid>
```

At the server level, a distribution descriptor downloads the specified directories for the private use of the server:

XML

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        <node name="Node">
            <server id="SimpleServer" ...>
                <distrib>
                    <directory>ServerFiles</directory>
                </distrib>
            </server>
        </node>
    </application>
</icegrid>
```

When a distribution descriptor is defined at both the application and server levels, as shown in the previous example, IceGrid assumes that a

dependency relationship exists between the two unless the server is configured otherwise. IceGrid checks this dependency before patching a server; if the server is dependent on the application's distribution, IceGrid patches the application's distribution first, and then proceeds to patch the server's. You can disable this dependency by modifying the server's descriptor:

XML

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        <node name="Node">
            <server id="SimpleServer" application-distrib="false" ...>
                <distrib>
                    <directory>ServerFiles</directory>
                </distrib>
            </server>
        </node>
    </application>
</icegrid>
```

Setting the `application-distrib` attribute to `false` informs IceGrid to consider the two distributions independent of one another.

Server Integrity during Distribution

Before an IceGrid node begins patching a distribution, it ensures that all relevant servers are shut down and prevents them from reactivating until patching completes. For example, the node disables all of the servers whose descriptors declare a dependency on the application distribution.

Distribution Descriptor Variables

The node stores application and server distributions in its data directory. The path names of the distributions are represented by reserved variables that you can use in your descriptors:

- `application.distrib`
  This variable can be used within server descriptors to refer to the top-level directory of the application distribution.
- `server.distrib`
  The value of this variable is the top-level directory of a server distribution. It can be used only within a server descriptor that has a distribution.

The XML example shown below illustrates the use of these variables:

XML

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        <node name="Node">
            <server id="Server1"
exe="${application.distrib}/Common/Bin/Server1" ...>
            </server>
            <server id="Server2"
exe="${server.distrib}/Server2Files/Bin/Server2" ...>
                <option>-d</option>
                <option>${server.distrib}/Server2Files</option>
                <distrib>
                    <directory>Server2Files</directory>
                </distrib>
            </server>
        </node>
    </application>
</icegrid>
```

Notice that the descriptor for `Server2` supplies the server's distribution directory as command-line options.

Using Distribution in the Ripper Application

Adding an application distribution to our ripper example requires two minor changes to our descriptors:

XML

```
<icegrid>
    <application name="Ripper">
        <replica-group id="EncoderAdapters">
            <load-balancing type="adaptive"/>
            <object identity="EncoderFactory"
type="::Ripper::MP3EncoderFactory"/>
        </replica-group>
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath" default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}" exe="${exepath}"
activation="on-demand">
                <adapter name="EncoderAdapter"
replica-group="EncoderAdapters"
                         endpoints="tcp"/>
            </server>
        </server-template>
        <distrib/>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
index="1"/>
            <server-instance template="IcePatch2"
directory="/opt/ripper/icepatch"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
index="2"/>
        </node>
    </application>
</icegrid>
```

An application distribution is sufficient for this example because we are deploying the same server on each node. We have also deployed an IcePatch2 server on `Node1` using the template.

See Also

- IcePatch2
- icegridadmin Command Line Tool
- IceGrid Templates
- Distrib Descriptor Element
- Using Descriptor Variables and Parameters
- Object Adapter Replication
- IcePatch2 Object Identities

IceGrid Administrative Sessions

To access IceGrid's administrative facilities from a program, you must first establish an administrative session. Once done, a wide range of services are at your disposal, including the manipulation of IceGrid registries, nodes, and servers; deployment of new components such as well-known objects; and dynamic monitoring of IceGrid events.

Note that, for replicated registries, an administrative session can be established with either the master or a slave registry replica, but a session with a slave replica is restricted to read-only operations.

On this page:

Creating an Administrative Session

The `Registry` interface provides two operations for creating an administrative session:

Slice

```
module IceGrid
{
    exception PermissionDeniedException
    {
        string reason;
    }

    interface Registry
    {
        AdminSession* createAdminSession(string userId, string password)
            throws PermissionDeniedException;

        AdminSession* createAdminSessionFromSecureConnection()
            throws PermissionDeniedException;

        idempotent int getSessionTimeout();

        idempotent int getACMTimeout();
        // ...
    }
}
```

The `createAdminSession` operation expects a username and password and returns a session proxy if the client is allowed to create a session. By default, IceGrid does not allow the creation of administrative sessions. You must define the property `IceGrid.Registry.AdminPermissionsVerifier` with the proxy of a permissions verifier object to enable session creation with `createAdminSession`. The verifier object must implement the interface `Glacier2::PermissionsVerifier`.

The `createAdminSessionFromSecureConnection` operation does not require a username and password because it uses the credentials supplied by an SSL connection to authenticate the client. As with `createAdminSession`, you must configure the proxy of a permissions verifier object before clients can use `createAdminSessionFromSecureConnection` to create a session. In this case, the `IceGrid.Registry.AdminSSLPermissionsVerifier` property specifies the proxy of a verifier object that implements the interface `Glacier2::SSLPermissionsVerifier`.

As an example, the following code demonstrates how to obtain a proxy for the registry and invoke `createAdminSession`:

C++11 C++98

```
auto base = communicator->stringToProxy("IceGrid/Registry");
auto registry = Ice::checkedCast<IceGrid::RegistryPrx>(base);
string username = ...;
string password = ...;
shared_ptr<IceGrid::AdminSessionPrx> session;
try
{
    session = registry->createAdminSession(username, password);
}
catch(const IceGrid::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
```

```
Ice::ObjectPrx base = communicator->stringToProxy("IceGrid/Registry");
IceGrid::RegistryPrx registry =
IceGrid::RegistryPrx::checkedCast(base);
string username = ...;
string password = ...;
IceGrid::AdminSessionPrx session;
try
{
    session = registry->createAdminSession(username, password);
}
catch(const IceGrid::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
```

The `AdminSession` interface provides operations for accessing log files and establishing observers. Furthermore, two additional operations are worthy of your attention:

Slice

```
module IceGrid
{
    interface AdminSession extends Glacier2::Session
    {
        idempotent void keepAlive();
        idempotent Admin* getAdmin();
        // ...
    }
}
```

If your program uses an administrative session indefinitely, you must prevent the session from expiring. You have two options for keeping a session alive:

1. Call `Registry::getSessionTimeout` and periodically invoke `AdminSession::keepAlive` every *timeout* (or less) seconds
2. Call `Registry::getACMTimeout` and configure ACM settings on the connection

We recommend using the second approach, which you can implement as follows:

C++11 C++98

```
int acmTimeout = registry->getACMTimeout();
if(acmTimeout > 0)
{
    auto conn = session->ice_getCachedConnection();
    conn->setACM(acmTimeout, Ice::nullopt,
Ice::ACMHeartbeat::HeartbeatAlways);
}
```

```
int acmTimeout = registry->getACMTimeout();
if(acmTimeout > 0)
{
    Ice::ConnectionPtr conn = session->ice_getCachedConnection();
    conn->setACM(acmTimeout, IceUtil::None, Ice::HeartbeatAlways);
}
```

Enabling heartbeats on the connection causes Ice to automatically send a heartbeat message at regular intervals determined by the given timeout value. The server ignores these messages, but they serve the purpose of keeping the session alive.

The `getAdmin` operation returns a proxy for the `IceGrid::Admin` interface, which provides complete access to the registry's settings. For this reason, you must use extreme caution when enabling administrative sessions.

Accessing Log Files Remotely

IceGrid's `AdminSession` interface provides operations for remotely accessing the log files of a registry, node, or server:

Slice

```
module IceGrid
{
    interface AdminSession extends Glacier2::Session
    {
        // ...
        FileIterator* openServerLog(string id, string path, int count)
            throws FileNotAvailableException, ServerNotExistException,
                   NodeUnreachableException, DeploymentException;
        FileIterator* openServerStdErr(string id, int count)
            throws FileNotAvailableException, ServerNotExistException,
                   NodeUnreachableException, DeploymentException;
        FileIterator* openServerStdOut(string id, int count)
            throws FileNotAvailableException, ServerNotExistException,
                   NodeUnreachableException, DeploymentException;
        FileIterator* openNodeStdErr(string name, int count)
            throws FileNotAvailableException, NodeNotExistException,
                   NodeUnreachableException;
        FileIterator* openNodeStdOut(string name, int count)
            throws FileNotAvailableException, NodeNotExistException,
                   NodeUnreachableException;
        FileIterator* openRegistryStdErr(string name, int count)
            throws FileNotAvailableException,
                   RegistryNotExistException,
                   RegistryUnreachableException;
        FileIterator * openRegistryStdOut(string name, int count)
            throws FileNotAvailableException,
                   RegistryNotExistException,
                   RegistryUnreachableException;
    }
}
```

In order to access the text of a program's standard output or standard error log, you must configure it using the `Ice.StdOut` and `Ice.StdErr` properties, respectively. For registries and nodes, you must define these properties explicitly but, for servers, the node defines these properties automatically if the property `IceGrid.Node.Output` is defined, causing the server's output to be logged in individual files.

If `IceGrid.Node.Output` is *not* defined, the following rules apply:

- If the node is started from a console or shell, servers share the node's `stdout` and `stderr`. If `Ice.StdOut` and/or `Ice.StdErr` properties are defined for the node, the servers' output is redirected to the specified files as well.
- If the node is started as a Unix daemon and `--noclose` is not used, the servers' output is lost, except if `Ice.StdOut` and/or `Ice.StdErr` properties are set for the node, in which case the servers' output is redirected to the specified files.
- If the node is started as a Windows service, the servers' output is lost even if `Ice.StdOut` and/or `Ice.StdErr` are set.

Log messages from the node itself are sent to `stderr` unless you set `Ice.UseSyslog` (for Unix). If the node is started as a Windows service, its log messages always are sent to the Windows event log.

In the case of `openServerLog`, the value of the `path` argument must resolve to the same file as one of the server's log descriptors. This security measure prevents a client from opening an arbitrary file on the server's host.

All of the operations accept a `count` argument and return a proxy to a `FileIterator` object. The `count` argument determines where to start reading the log file: if the value is negative, the iterator is positioned at the beginning of the file, otherwise the iterator is positioned to return the last `count` lines of text.

The `FileIterator` interface is quite simple:

Slice

```
module IceGrid
{
    interface FileIterator
    {
        bool read(int size, out Ice::StringSeq lines)
            throws FileNotAvailableException;
        void destroy();
    }
}
```

A client may invoke the `read` operation as many times as necessary. The `size` argument specifies the maximum number of bytes that `read` can return; the client must not use a size that would cause the reply to exceed the client's configured maximum message size.

If this is the client's first call to `read`, the `lines` argument holds whatever text was available from the iterator's initial position, and the iterator is repositioned in preparation for the next call to `read`. The operation returns false to indicate that more text is available and true if all available text has been read.

Line termination characters are removed from the contents of `lines`. When displaying the text, you must be aware that the first and last elements of the sequence can be partial lines. For example, the last line of the sequence might be incomplete if the limit specified by `size` is reached. The next call to `read` returns the remainder of that line as the first element in the sequence.

As an example, the C++ code below displays the contents of a log file and waits for new text to become available:

C++11 C++98

```
shared_ptr<IceGrid::FileIteratorPrx> iter = ...;
while(true)
{
    Ice::StringSeq lines;
    bool end = iter->read(10000, lines);
    if(!lines.empty())
    {
        // The first line might be a continuation from
        // the previous call to read.
        cout << lines[0];
        for(const auto& p : lines)
        {
            cout << endl << p << flush;
        }
    }
    if(end)
    {
        sleep(1);
    }
}
```

```
IceGrid::FileIteratorPrx iter = ...;
while(true)
{
    Ice::StringSeq lines;
    bool end = iter->read(10000, lines);
    if(!lines.empty())
    {
        // The first line might be a continuation from
        // the previous call to read.
        cout << lines[0];
        for(Ice::StringSeq::const_iterator p =
++lines.begin(); p != lines.end(); ++p)
        {
            cout << endl << *p << flush;
        }
    }
    if(end)
    {
        sleep(1);
    }
}
```

Notice that the loop includes a delay in case `read` returns true, which prevents the client from entering a busy loop when no data is currently available.

The client should call `destroy` when the iterator object is no longer required. At the time the client's session terminates, IceGrid reclaims any iterators that were not explicitly destroyed.

If the client waits for new data, it must take steps to prevent the administrative session from expiring.

With these operations, an administrative client can retrieve any text file on a system where an IceGrid node is running. While it's common for this text file to contain the output of an Ice logger, it could contain other unrelated outputs. This text file is presented as a sequence of strings, with no particular structure for these strings.

The `Logger` admin facet provides another, and often better, way to retrieve the log messages sent to the logger of a server, node or registry. With the `Logger` facet, you retrieve log messages–and only log messages–as typed structures, whether or not the logger's output is stored in a text file or stored at all. With the `AdminSession` file operations presented above, you can retrieve the log files of a server even when this server is not running; conversely, with the `Logger` admin facet, the target server must be running since this `Logger` facet is hosted in that server. Finally, the programming style espoused by the `AdminSession` file operations is a pull model: the administrative client calls `read` from time to time on the `FileIterator` object provided by IceGrid. With the `Logger` admin facet, the administrative client uses a push model: it registers a remote logger object with the target server, node and registry, and this remote logger receives new log messages as soon as they are generated.

Dynamic Monitoring in IceGrid

IceGrid allows an application to monitor relevant state changes by registering callback objects. (The IceGrid GUI tool uses these callback interfaces for its implementation.) The callback interfaces are useful to, for example, automatically generate an email notification when a node goes down or some other state change of interest occurs.

Observer Interfaces

IceGrid offers a callback interface for each major component of the IceGrid architecture:

Slice

```
module IceGrid
{
    interface NodeObserver
    {
        void nodeInit(NodeDynamicInfoSeq nodes);
        void nodeUp(NodeDynamicInfo node);
        void nodeDown(string name);
        void updateServer(string node, ServerDynamicInfo updatedInfo);
        void updateAdapter(string node, AdapterDynamicInfo updatedInfo);
    }

    interface ApplicationObserver
    {
        void applicationInit(int serial,
ApplicationInfoSeq applications);
        void applicationAdded(int serial, ApplicationInfo desc);
        void applicationRemoved(int serial, string name);
        void applicationUpdated(int serial, ApplicationUpdateInfo desc);
    }

    interface AdapterObserver
    {
        void adapterInit(AdapterInfoSeq adpts);
        void adapterAdded(AdapterInfo info);
        void adapterUpdated(AdapterInfo info);
        void adapterRemoved(string id);
    }

    interface ObjectObserver
    {
        void objectInit(ObjectInfoSeq objects);
        void objectAdded(ObjectInfo info);
        void objectUpdated(ObjectInfo info);
        void objectRemoved(Ice::Identity id);
    }

    interface RegistryObserver
    {
        void registryInit(RegistryInfoSeq registries);
        void registryUp(RegistryInfo node);
        void registryDown(string name);
    }
}
```

The next section describes how to install an observer.

Registering Observers

The `AdminSession` interface provides two operations for registering your observers:

Slice

```
module IceGrid
{
    interface AdminSession extends Glacier2::Session
    {
        idempotent void keepAlive();

        idempotent void setObservers(RegistryObserver* registryObs,
                                     NodeObserver* nodeObs,
                                     ApplicationObserver* appObs,
                                     AdapterObserver* adptObs,
                                     ObjectObserver* objObs)
            throws ObserverAlreadyRegisteredException;


 idempotent void setObserversByIdentity(Ice::Identity registryObs,
                                        Ice::Identity nodeObs,
                                        Ice::Identity appObs,
                                        Ice::Identity adptObs,
                                        Ice::Identity objObs)
            throws ObserverAlreadyRegisteredException;
        // ...
    }
}
```

You should invoke `setObservers` and supply proxies when it is possible for the registry to establish a separate connection to the client to deliver its callbacks. If network restrictions such as firewalls prevent such a connection, you should use the `setObserversByIdentity` operation, which creates a bidirectional connection instead.

You can pass a null proxy for any parameter to `setObservers`, or an empty identity for any parameter to `setObserversByIdentity`, if you want to use only some of the observers. In addition, passing a null proxy or an empty identity for an observer cancels a previous registration of that observer. The operations raise `ObserverAlreadyRegisteredException` if you pass a proxy or identity that was registered in a previous call.

Once the observers are registered, operations corresponding to state changes will be invoked on the observers. (See the Slice API Reference for details on the data passed to the observers. You can also look at the source code for the IceGrid GUI implementation in the Ice for Java distribution to see how observers are used by the GUI.)

Finally, remember to take the steps necessary to prevent the administrative session from expiring.

See Also

- Registry Replication
- Securing a Glacier2 Router
- Resource Allocation using IceGrid Sessions
- Log Descriptor Element
- icegridadmin Command Line Tool
- Bidirectional Connections
- IceGrid.*
- Logger admin Facet

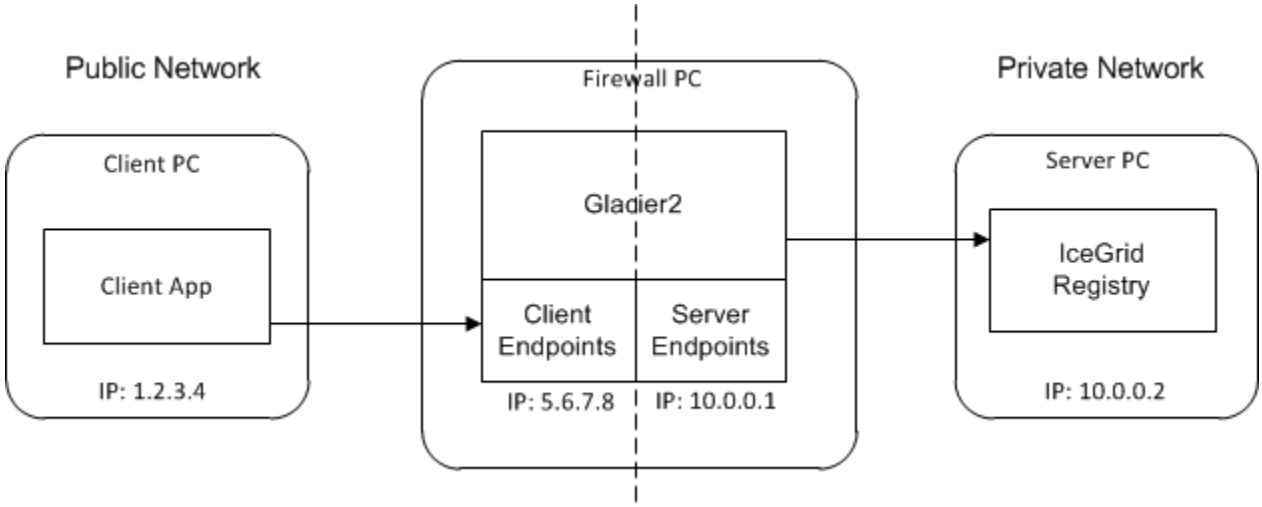
Glacier2 Integration with IceGrid

This section provides information on integrating a Glacier2 router into your IceGrid environment.

On this page:

Configuration Changes for using Glacier2 with IceGrid

A typical IceGrid client must be configured with a locator proxy, but the configuration requirements change when the client accesses the location service indirectly via a Glacier2 router as shown below:



Using IceGrid via a Glacier2 router.

In this situation, it is the router that must be configured with a locator proxy.

Assuming the registry's client endpoint in the illustration uses port `8000`, the router requires the following setting for the `Ice.Default.Locator` property:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h 10.0.0.2 -p 8000
```

Fortunately, the node supplies this property when it starts the router, so there is no need to configure it explicitly. Note that all of the router's clients use the same locator.

Remote IceGrid Administration via Glacier2

If you intend to administer IceGrid remotely via a Glacier2 router, you must define one of the following properties (or both), depending on whether you use user name and password authentication or a secure connection:

```
Glacier2.SessionManager=IceGrid/AdminSessionManager
Glacier2.SSLSessionManager=IceGrid/AdminSSLSessionManager
```

These session managers are accessible via the registry's administrative session manager endpoints, so the Glacier2 router must be authorized to establish a connection to these endpoints. Note that you must secure these endpoints, otherwise arbitrary clients can manipulate the session managers. An administrative session is allowed to access any object by default. To restrict access to the `IceGrid::AdminSession` object and the `IceGrid::Admin` object that is returned by the session's `getAdmin` operation, you must set the property `IceGrid.Registry.AdminSessionFilters` to one.

Resource Allocation using Glacier2 and IceGrid

To allocate servers and objects, a program can establish a client session via Glacier2. Depending on the authentication method, one or both of the following properties must be set in the Glacier2 configuration:

```
Glacier2.SessionManager=IceGrid/SessionManager
Glacier2.SSLSessionManager=IceGrid/SSLSessionManager
```

These session managers are accessible via the registry's session manager endpoints, so the Glacier2 router must be authorized to establish a connection to these endpoints.

A client session is allowed to access any object by default. To restrict access to the `IceGrid::Session` and `IceGrid::Query` objects, you must set the property `IceGrid.Registry.SessionFilters` to one. However, you can use the allocation mechanism to access additional objects and adapters. IceGrid adds an identity filter when a client allocates an object and removes that filter again when the object is released. When a client allocates a server, IceGrid adds an adapter identity filter for the server's indirect adapters and removes that filter again when the server is released.

Session Considerations for Glacier2 and IceGrid

Providing access to administrative sessions and client sessions both require that you define at least one of the properties `Glacier2.SessionManager` and `Glacier2.SSLSessionManager`, which presents a potential problem if you intend to access both types of sessions via the same Glacier2 router.

The simplest solution is to dedicate a router instance to each type of session. However, if you need to access both types of sessions from a single router, you can accomplish it only if you use a different authentication mechanism for each type of session. For example, you can configure the router as follows:

```
Glacier2.SessionManager=IceGrid/SessionManager
Glacier2.SSLSessionManager=IceGrid/AdminSSLSessionManager
```

This configuration uses user name and password authentication for client sessions, and SSL authentication for administrative sessions. If this restriction is too limiting, you must use two router instances.

Deploying Glacier2 with IceGrid

The Ice distribution includes default server templates for Ice services such as IcePatch2 and Glacier2 that simplify the task of deploying these servers in an IceGrid domain.

The relevant portion from the file `config/template.xml` is shown below:

XML

```
<server-template id="Glacier2">
    <parameter name="instance-name" default="${application}.Glacier2"/>
    <parameter name="client-endpoints"/>
    <parameter name="server-endpoints"/>
    <parameter name="session-timeout" default="0"/>

    <server id="${instance-name}" exe="glacier2router">
    <properties>
        <property name="Glacier2.Client.Endpoints"
value="${client-endpoints}"/>
        <property name="Glacier2.Server.Endpoints"
value="${server-endpoints}"/>
        <property name="Glacier2.InstanceName"
value="${instance-name}"/>
        <property name="Glacier2.SessionTimeout"
value="${session-timeout}"/>
    </properties>
</server-template>
```

Notice that the server's pathname is `glacier2router`, meaning the program must be present in the node's executable search path. Another important point is the server's activation mode: it uses manual activation (the default), meaning the router must be started manually. This requirement becomes clear when you consider that the router is the point of contact for remote clients; if the router is not running, there is no way for a client to contact the locator and cause the router to be started on-demand.

The template defines only a few properties; if you want to set additional properties, you can define them in the server instance property set.

Of interest is the `instance-name` parameter, which allows you to configure the `Glacier2.InstanceName` property. The parameter's default value includes the name of the application in which the template is used. This parameter also affects the identities of the objects implemented by the router.

Consider the following sample application:

```
<icegrid>
    <application name="Glacier2Demo">
        <node name="Node">
            <server-instance template="Glacier2"
                client-endpoints="tcp -h 5.6.7.8 -p 8000"
                session-timeout="300"
                server-endpoints="tcp -h 10.0.0.1"/>
            ...
        </node>
    </application>
</icegrid>
```

Instantiating the `Glacier2` template creates a server identified as `Glacier2Demo.Glacier2` (as determined by the default value for the `instance-name` parameter). The router's objects use this value as the category in their identities, such as `Glacier2Demo.Glacier2/router`. The router proxy used by clients must contain a matching identity.

In order to refer to the `Glacier2` template in your application, you must have already configured the registry to use the `config/templates.xml` file as your default templates, or copied the template into the XML file describing your application.

Note that IceGrid cannot start a Glacier2 router if the router's security configuration requires that a passphrase be entered. In this situation, you have no choice but to start the router yourself so that you can provide the passphrase when prompted.

See Also

- Glacier2
- IceGrid Templates
- Getting Started with Glacier2
- Glacier2.*
- IceGrid.*

IceGrid XML Reference

This section provides a reference for the XML elements that define IceGrid descriptors, in alphabetical order.

IceGrid XML files must use UTF-8 encoding.

Topics

See Also

- Using IceGrid Deployment

Adapter Descriptor Element

An `adapter` element defines an indirect object adapter.

This element may only appear as a child of a `server` or `service` element.

The following attributes are supported:

Attribute	Description	Required
`endpoints`	Specifies one or more endpoints for this object adapter. These endpoints typically do not specify a port. This attribute is translated into a definition of the adapter's `Endpoints` configuration property	No
`id`	Specifies an object adapter identifier. The identifier must be unique among all adapters and replica groups in the registry. This attribute is translated into a definition of the adapter's `AdapterId` configuration property. If not defined, a default value is constructed from the adapter name and server ID (and service name for an IceBox service).	Yes
`name`	The name of the object adapter as used in the server that creates it.	Yes
`priority`	Specifies the priority of the object adapter as an integer value. The object adapter priority is used by the `Ordered` replica group load balancing policy to determine the order of the endpoints returned by a locate request. Endpoints are ordered from the smallest priority value to the highest. If not defined, the value is 0.	No
`proxy-options`	The proxy options to use when generating the adapter well-known or allocatable object proxies. This attribute is also translated into a definition of the adapter's ProxyOptions configuration property.	No
`register-process`	This attribute is only valid if the enclosing server uses an Ice version prior to 3.3. In Ice 3.3 or later, this functionality is replaced by the administrative facility. If the value is `true`, this object adapter registers an object in the IceGrid registry that facilitates graceful shutdown of the server. Only one object adapter in a server should set this attribute to `true`. If not defined, the default value is `false`.	No
`replica-group`	Specifies a replica group identifier. A non-empty value signals that this object adapter is a member of the indicated replica group. This attribute is translated into a definition of the adapter's `ReplicaGroupId` configuration property. If not defined, the default value is an empty string.	No
`server-lifetime`	A value of `true` indicates that the lifetime of this object adapter is the same as the lifetime of its server. This information is used by the IceGrid node to determine the state of the server. Specifically, the server is considered activated when all the object adapters with the `server-lifetime` attribute set to `true` are registered with the registry (the object adapter registers itself during activation). Furthermore, server deactivation is considered to begin as soon as one object adapter with the `server-lifetime` attribute set to `true` is unregistered with the registry (the object adapter unregisters itself during deactivation).	No

An optional nested `description` element provides free-form descriptive text.

Here is an example to demonstrate the use of this element.

XML

```
<adapter name="MyAdapter"
    endpoints="default"
    id="MyAdapterId"
    proxy-options="-t -e 1.0"
    replica-group="MyReplicaGroup">
    <description>A description of the adapter.</description>
    ...
</adapter>
```

See Also

Allocatable Descriptor Element

An allocatable element defines an allocatable object in the IceGrid registry. Clients can allocate this object using only its identity, or they can allocate objects of a specific type.

This element may only appear as a child of an `adapter` element.

The following attributes are supported:

Attribute	Description	Required
`identity`	Specifies the identity by which this allocatable object is known.	Yes
`property`	Specifies the name of a property to generate that contains the stringified identity of this allocatable.	No
`type`	An arbitrary string used to group allocatable objects. By convention, the string represents the most-derived Slice type ID of the object, but an application is free to use another convention.	No
`proxy-options`	The proxy options to use for the proxy of the allocatable object returned by IceGrid.	No

See Also

- Resource Allocation using IceGrid Sessions
- Adapter Descriptor Element
- Type IDs

Application Descriptor Element

An `application` element defines an IceGrid application. An application typically contains at least one `node` element, but it may also be used for other purposes such as defining server and service templates, default templates, replica groups and property sets.

This element must be a child of an `icegrid` element. Only one `application` element is permitted per file.

The following attributes are supported.

Attribute	Description	Required
`import-default-templates`	If `true`, the default templates configured for the IceGrid registry are imported and available for use within this application. If not specified, the default value is `false`.	No
`name`	The name of the application. This name must be unique among all applications in the registry. Within the application, child elements can refer to its name using the reserved variable `${application}`.	Yes

An optional nested `description` element provides free-form descriptive text.

Here is an example to demonstrate the use of this element:

XML

```
<icegrid>
    <application name="MyApplication" import-default-templates="true">
        <description>A description of the application.</description>
        ...
    </application>
</icegrid>
```

See Also

- IceGrid Architecture
- Node Descriptor Element
- Server-Template Descriptor Element
- Service-Template Descriptor Element
- Replica-Group Descriptor Element
- Properties Descriptor Element
- IceGrid Descriptor Element
- Description Descriptor Element
- IceGrid Templates
- Using Descriptor Variables and Parameters

DbEnv Descriptor Element

A `dbenv` element causes an IceGrid node to generate Freeze configuration properties for the server or service in which it is defined. The IceGrid node also creates a directory for the Berkeley DB database environment if needed.

This element may contain `dbproperty` elements.This element may only appear as a child of a `server` or `service` element.

The following attributes are supported:

Attribute	Description	Required
`home`	Specifies the directory to use as the database environment. If not defined, a sub-directory in the node's data directory is used.	No
`name`	The name of the Berkeley DB database environment.	Yes

The values of the `name` and `home` attributes are used to define the `Freeze.DbEnv.`*`env-name`*`.DbHome` property as shown below:

`Freeze.DbEnv.`*`name`*`.DbHome=`*`home`*

An optional nested `description` element provides free-form descriptive text.

Here is an example to demonstrate the use of this element:

XML

```
<dbenv name="MyEnv" home="/opt/data/db">
    <description>A description of the Freeze/BDB
database environment.</description>
    ...
</dbenv>
```

See Also

- Freeze Manual
- DbProperty Descriptor Element
- Server Descriptor Element
- Service Descriptor Element
- Description Descriptor Element

DbProperty Descriptor Element

A `dbproperty` element defines a Berkeley DB configuration property for a Freeze database environment.

This element may only appear as a child of a `dbenv` element.

The following attributes are supported:

Attribute	Description	Required
`name`	The name of the configuration property.	Yes
`value`	The value of the configuration property. If not defined, the value is an empty string.	No

Here is an example to demonstrate the use of this element:

XML

```
<dbenv name="MyEnv" home="/opt/data/db">
    <description>A description of the
database environment.</description>
    <dbproperty name="set_cachesize" value="0 52428800 1"/>
</dbenv>
```

See Also

- Freeze Manual
- DbEnv Descriptor Element

Description Descriptor Element

A `description` element specifies a description of its parent element.

This element may only appear as a child of the `application`, `replica-group`, `node`, `server`, `service`, `icebox`, `adapter`, and `dbenv` elements.

Here is an example to demonstrate the use of this element:

XML

```
<node name="localnode">
    <description>Free form descriptive text for localnode</description>
</node>
```

See Also

- Application Descriptor Element
- Replica-Group Descriptor Element
- Node Descriptor Element
- Server Descriptor Element
- Service Descriptor Element
- IceBox Descriptor Element
- Adapter Descriptor Element
- DbEnv Descriptor Element

Directory Descriptor Element

A `directory` element specifies a directory in a distribution.

This element may only appear as a child of the `distrib` element.

This element supports no attributes.

See Also

- Application Distribution
- Distrib Descriptor Element

Distrib Descriptor Element

A `distrib` element specifies the distribution files to download from an IcePatch2 server as well as the server's proxy.

This element may only appear as a child of an `application` element or a `server` element.

The following attributes are supported:

Attribute	Description	Required
`icepatch`	Proxy for the IcePatch2 server. If not defined, the default value is `${application}.IcePatch2/server`.	No

Here is an example to demonstrate the use of this element:

XML

```
<distrib icepatch="DemoIcePatch2/server:tcp -p 12345">
    <directory>dir1</directory>
    <directory>dir2/subdir</directory>
</distrib>
```

See Also

- Application Distribution
- IcePatch2
- Application Descriptor Element
- Server Descriptor Element

IceBox Descriptor Element

An `icebox` element defines an IceBox server to be deployed on a node. It typically contains at least one `service` element, and may supply additional information such as command-line options, environment variables, configuration properties and a server distribution.

This element may only appear as a child of a `node` element or a `server-template` element.

This element supports the same attributes as the `server` element.

An optional nested `description` element provides free-form descriptive text.

Here is an example to demonstrate the use of this element:

XML

```
<icebox id="MyIceBox"
        activation="on-demand"
        activation-timeout="60"
        application-distrib="false"
        deactivation-timeout="60"
        exe="/opt/Ice/bin/icebox"
        pwd="/">
    <option>--Ice.Trace.Network=1</option>
    <env>PATH=/opt/Ice/bin:$PATH</env>
    <property name="IceBox.UseSharedCommunicator.Service1" value="1"/>
    <service name="Service1" .../>
    <service-instance template="ServiceTemplate" name="Service2"/>
</icebox>
```

See Also

- IceBox
- Service Descriptor Element
- Adapter Descriptor Element
- Properties Descriptor Element
- Node Descriptor Element
- Server-Template Descriptor Element
- Server Descriptor Element
- Description Descriptor Element
- Using Command Line Options in Descriptors
- Setting Environment Variables in Descriptors
- Application Distribution
- Administrative Facility

IceGrid Descriptor Element

The `icegrid` element is the top-level element for IceGrid descriptors in XML files. This elements supports no attributes.

See Also

- Using IceGrid Deployment

Load-Balancing Descriptor Element

A `load-balancing` element determines the load balancing policy used by a replica group.

This element may only appear as a child of a `replica-group` element.

The following attributes are supported:

Attribute	Description	Required
`load-sample`	Specifies the load sample to use for the adaptive load balancing policy. It can only be defined if `type` is set to `adaptive`. Legal values are `1`, `5` and `15`. If not specified, the load sample default value is `1`.	No
`n-replicas`	Specifies the maximum number of replicas used to compute the endpoints of the replica group. If not specified, the default value is `1`.	No
`type`	Specifies the type of load balancing. Legal values are `adaptive`, `ordered`, `round-robin` and `random`.	Yes

Here is an example to demonstrate the use of this element:

XML

```
<application name="MyApp">
    <replica-group id="ReplicatedAdapter">
        <load-balancing type="adaptive" load-sample="15"
n-replicas="3"/>
        <description>A description of this replica group.</description>
        <object identity="WellKnownObject" .../>
    </replica-group>
    ...
</application>
```

See Also

- Load Balancing
- Object Adapter Replication
- Replica-Group Descriptor Element

Log Descriptor Element

A `log` element specifies the name of a log file for a server or service. A `log` element must be defined for each log file that can be accessed remotely by an administrative tool. Note that it is not necessary to define a `log` element for the values of the `Ice.StdErr` and `Ice.StdOut` properties.

This element may only appear as a child of a `server` element or a `service` element.

The following attributes are supported:

Attribute	Description	Required
`path`	The path name of the log file. If a relative path is specified, it is relative to the current working directory of the node. The node must have sufficient access privileges to read the file.	Yes
`property`	Specifies the name of a property in which to store the path name of the log file as given in the `path` attribute.	No

Here is an example to demonstrate the use of this element:

XML

```
<server id="MyServer" ...>
    <log path="${server}.log" property="LogFile"/>
</server>
```

See Also

- IceGrid Administrative Sessions
- Server Descriptor Element
- Service Descriptor Element

Node Descriptor Element

A `node` element defines an IceGrid node. The servers that the node is responsible for managing are described in child elements.

This element may only appear as a child of an `application` element. Multiple `node` elements having the same name may appear in an application. Their contents are merged and the last definition of `load-factor` has precedence.

The following attributes are supported:

Attribute	Description	Required
`load-factor`	A floating point value defining the factor that is multiplied with the node's load average. The load average is used by the adaptive load balancing policy to figure out which node is the least loaded. The default is `1.0` on Unix platforms and `1/NCPUS` on Windows (where `NCPUS` is the number of CPUs in the node's computer). Note that, if Unix and Windows machines are part of a replica group, the Unix and Windows figures are not directly comparable, but the registry still makes an attempt to pick the least-loaded node.	No
`name`	The name must be unique among all nodes in the registry. Within the node, child elements can refer to its name using the reserved variable `${node}`. An `icegridnode` process representing this node must be started on the desired computer and its configuration property `IceGrid.Node.Name` must match this attribute.	Yes

Here is an example to demonstrate the use of this element:

XML

```
<node name="Node1" load-factor="2.0">
    <description>A description of this node.</description>
    <server id="Server1" ...>
        <property name="NodeName" value="${node}"/>
        ...
    </server>
</node>
```

See Also

- Application Descriptor Element
- Load Balancing
- Object Adapter Replication
- icegridnode

Object Descriptor Element

An `object` element defines a well-known object in the IceGrid registry. Clients can refer to this object using only its identity, or they can search for well-known objects of a specific type.

This element may only appear as a child of an `adapter` element or a `replica-group` element.

The following attributes are supported:

Attribute	Description	Required
`identity`	Specifies the identity by which this object is known.	Yes
`property`	Specifies the name of a property to generate that contains the stringified identity of the object. This attribute is only allowed if this `object` element is a child of an `adapter` element.	No
`type`	An arbitrary string used to group objects. By convention, the string represents the most-derived Slice type ID of the object, but an application is free to use another convention.	No
`proxy-options`	The proxy options to use for the proxy of the well-known object return by IceGrid.	No

Here is an example to demonstrate the use of this element:

XML

```
<adapter name="MyAdapter" id="WellKnownAdapter" ...>
    <object identity="WellKnownObject"
type="::Module::WellKnownInterface" proxy-options="-o"/>
</adapter>
```

In the configuration above, the object can be located via the equivalent proxies `WellKnownObject` and `WellKnownObject@WellKnownAdapter`.

See Also

- Well-Known Objects
- Adapter Descriptor Element
- Replica-Group Descriptor Element
- Type IDs

Parameter Descriptor Element

A `parameter` element defines a template parameter. Template parameters must be declared with this element to be used in template instantiation.

This element may only appear as a child of a `server-template` element or a `service-template` element.

The following attributes are supported:

Attribute	Description	Required
`name`	The name of the parameter. For example, if `index` is used as the name of a parameter, it can be referenced using `${index}` in the server or service template.	Yes
`default`	An optional default value for the parameter. This value is used if the parameter is not defined when a server or service is instantiated.	No

Here is an example to demonstrate the use of this element:

XML

```
<server-template id="MyServerTemplate">
    <parameter name="index"/>
    <parameter name="exepath" default="/opt/myapp/bin/server"/>
    ...
</server-template>
```

See Also

- IceGrid Templates
- Server-Template Descriptor Element
- Service-Template Descriptor Element
- Using Descriptor Variables and Parameters

Properties Descriptor Element

The `properties` element is used in three situations:

- as a named property set if the `id` attribute is specified
- as a reference to a named property set if the `refid` attribute is specified
- as an unnamed property set if the `id` or `refid` attributes are not specified.

A property set is useful for defining a set of properties (a named property set) in application or node descriptors. Named property sets can be included in named or unnamed property sets with property set references.

A named property set element may only be a child of an `application` element or a `node` element. An unnamed property set element may only be a child of a `server`, `icebox`, `service`, `server-instance` or `service-instance` element. An unnamed property set element with the `service` attribute defined may only be a child of a `server-instance` element. A reference to a named property set can only be a child of a named or unnamed property set element.

The following attributes are supported:

Attribute	Description	Required
`id`	Defines a new named property set with the given identifier. The identifier must be unique among all named property sets defined in the same scope. If not specified, the properties element refers to an unnamed property set or a property set reference.	No
`refid`	Defines a reference to the named property set with the given identifier. If not specified, the element refers to an unnamed or named property set.	No
`service`	Specifies the name of an IceBox service that is defined in the enclosing `server-instance` descriptor. The server instance must be an IceBox server that includes a service with the given name. An unnamed property set with this attribute defined extends the properties of the service. If not specified, the unnamed property set extends the properties of the server instance.	No

Here is an example to demonstrate the use of this element:

XML

```
<application name="Simple">
   <properties id="Debug">
        <property name="Ice.Trace.Network" value="1"/>
   </properties>

    <server id="MyServer" exe="./server">
        <properties>
            <properties refid="Debug"/>
            <property name="AppProperty" value="1"/>
        </properties>
    </server>
</application>
```

See Also

- Properties and Configuration
- Application Descriptor Element
- Node Descriptor Element
- Server Descriptor Element
- IceBox Descriptor Element
- Service Descriptor Element
- Server-Instance Descriptor Element
- Service-Instance Descriptor Element

Property Descriptor Element

An IceGrid node generates a configuration file for each of its servers and services. This file generally should not be edited manually because any changes are lost the next time the node generates the file. The `property` element is the correct way to define additional properties in a configuration file.

Note that IceGrid administrative utilities can retrieve the configuration properties of a server or service via the administrative facility.

This element may only appear as a child of a `server` element, a `service` element, an `icebox` element or a `properties` element.

The following attributes are supported:

Attribute	Description	Required
`name`	Specifies the property name.	Yes
`value`	Specifies the property value. If not defined, the value is an empty string.	No

Here is an example to demonstrate the use of this element:

XML

```
<server id="MyServer" ...>
    <property name="Ice.ThreadPool.Server.SizeMax" value="10"/>
    ...
</server>
```

This `property` element adds the following definition to the server's configuration file:

```
Ice.ThreadPool.Server.SizeMax=10
```

See Also

- Properties and Configuration
- icegridadmin Command Line Tool
- IceGrid and the Administrative Facility
- Server Descriptor Element
- Service Descriptor Element
- IceBox Descriptor Element
- Properties Descriptor Element

Replica-Group Descriptor Element

A `replica-group` element creates a virtual object adapter in order to provide replication and load balancing for a collection of object adapters. An `adapter` element declares its membership in a group by identifying the desired replica group. The element may declare well-known objects that are available in all of the participating object adapters. A `replica-group` element may contain a `load-balancing` child element that specifies the load-balancing algorithm the registry should use when resolving locate requests. If not specified, the registry uses a random load balancing policy with the number of replicas set to 0.

This element may only appear as a child of an `application` element.

The following attributes are supported:

Attribute	Description	Required
`id`	Specifies the identifier of the replica group, which must be unique among all adapters and replica groups in the registry. This identifier can be used in indirect proxies in place of an adapter identifier.	Yes
`filter`	Specifies the identifier of a replica group filter that performs custom load balancing.	No
`proxy-options`	The proxy options to use for well-known object proxies declared with this replica group.	No

An optional nested `description` element provides free-form descriptive text.

Here is an example to demonstrate the use of this element:

XML

```
<application name="MyApp">
    <replica-group id="ReplicatedAdapter" filter="myCustomFilter"
proxy-options="-e 1.0">
        <load-balancing type="adaptive" load-sample="15"
n-replicas="3"/>
        <description>A description of this replica group.</description>
        <object identity="WellKnownObject" .../>
    </replica-group>
    ...
</application>
```

In this example, the proxy `WellKnownObject` is equivalent to the proxy `WellKnownObject@ReplicatedAdapter`.

See Also

- Object Adapter Replication
- Load Balancing
- Adapter Descriptor Element
- Application Descriptor Element
- Description Descriptor Element
- Well-Known Objects

Server Descriptor Element

A `server` element defines a server to be deployed on a node. It typically contains at least one `adapter` element, and may supply additional information such as command-line options, environment variables, configuration properties, and a server distribution.

This element may only appear as a child of a `node` element or a `server-template` element.

The following attributes are supported:

Attribute	Description	Required
`activation`	Specifies whether the server's activation mode. Legal values are `manual`, `on-demand`, `always` and `session`. If not specified, manual activation is used by default.	No
`activation-timeout`	Defines the number of seconds a node will wait for the server to activate. If the timeout expires, a client waiting to receive the endpoints of an object adapter in this server will receive an empty set of endpoints. If not defined, the default timeout is the value of the `IceGrid.Node.WaitTime` property configured for the server's node.	No
`allocatable`	Specifies whether the server can be allocated. A server is allocated implicitly when one of its allocatable objects is allocated. The value of this attribute is ignored if the server activation mode is `session`; a server with this activation mode is always allocatable. Otherwise, if not specified and the server activation mode is not `session`, the server is not allocatable.	No
`application-distrib`	Specifies whether this server's distribution is dependent on the application's distribution. If the value is `true`, the server cannot be patched until the application has been patched. If not defined, the default value is `true`.	No
`deactivation-timeout`	Defines the number of seconds a node will wait for the server to deactivate. If the timeout expires, the node terminates the server process. If not defined, the default timeout is the value of the node's configuration property `IceGrid.Node.WaitTime`.	No
`exe`	The pathname of the server executable. On Windows, if a relative path is specified, the executable is searched in the system path. On Linux, it's searched in the user path specified by the PATH environment variable from the environment where the `icegridnode` is started.	Yes
`ice-version`	Specifies the Ice version in use by this server. If not defined, IceGrid assumes the server uses the same version that IceGrid itself uses. A server that uses an Ice version prior to 3.3 must define this attribute if its adapters use the `register-process` attribute. For example, a server using Ice 3.2.x should use `3.2` as the value of this attribute.	No
`id`	Specifies the identifier for this server. The identifier must be unique among all servers in the registry. Within the server, child elements can refer to its identifier using the reserved variable `${server}`.	Yes
`pwd`	The default working directory for the server. If not defined, the server is started in the node's current working directory.	No
`user`	Specifies the name of the user account under which the server is activated and run. If a user account mapper is configured for the node, the value of this attribute is used to find the corresponding account in the map. Node running on Windows as any user, or Linux and macOS as non-root: • The only permissible value for this attribute is an empty string or the name of the user account under which the node is running. Node running on Linux and macOS as root: • The node must be running as root to be able to run servers under a different user account. • If the node is running as root and this attribute is not specified, the server is run under the user `${session.id}` if the server activation mode is `session` or under the user `nobody` if the activation mode is `manual`, `on-demand` or `always`. • IceGrid.Node.AllowRunningServersAsRoot must be set to a non-zero value to allow running servers as root.	No

An optional nested `description` element provides free-form descriptive text.

Here is an example to demonstrate the use of this element:

XML

```
<server id="MyServer"
        activation="on-demand"
        activation-timeout="60"
        application-distrib="false"
        deactivation-timeout="60"
        exe="/opt/app/bin/myserver"
        pwd="/">
    <option>--Ice.Trace.Network=1</option>
    <env>PATH=/opt/Ice/bin:$PATH</env>
    <property name="ServerId" value="${server}"/>
    <adapter name="Adapter1" .../>
</server>
```

See Also

- Adapter Descriptor Element
- Properties Descriptor Element
- Node Descriptor Element
- Server-Template Descriptor Element
- Description Descriptor Element
- Using Command Line Options in Descriptors
- Setting Environment Variables in Descriptors
- Application Distribution
- IceGrid Server Activation
- Using Descriptor Variables and Parameters

Server-Instance Descriptor Element

A `server-instance` element deploys an instance of a `server-template` element on a node. It may supply additional information such as configuration properties.

This element may only appear as a child of a `node` element.

The following attributes are supported:

Attribute	Description	Required
`template`	Identifies the server template.	Yes

All other attributes of the element must correspond to parameters declared by the template. The `server-instance` element must provide a value for each parameter that does not have a default value supplied by the template.

Here is an example to demonstrate the use of this element:

XML

```
<icegrid>
    <application name="SampleApp">
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <server id="${id}" activation="manual" .../>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate" id="TheServer">
                <properties>
                    <property name="Debug" value="1"/>
                </properties>
            </server-instance>
        </node>
    </application>
</icegrid>
```

See Also

- Server-Template Descriptor Element
- Node Descriptor Element
- IceGrid Templates
- Using Descriptor Variables and Parameters

Server-Template Descriptor Element

A `server-template` element defines a template for a `server` element, simplifying the task of deploying multiple instances of the same server definition. The template should contain a parameterized `server` element that is instantiated using a `server-instance` element.

This element may only appear as a child of an `application` element.

The following attributes are supported:

Attribute	Description	Required
`id`	Specifies the identifier for the server template. This identifier must be unique among all server templates in the application.	Yes

A template may declare parameters that are used to instantiate the `server` element. You can define a default value for each parameter. Parameters without a default value are considered mandatory and values for them must be supplied by the `server-instance` element.

Here is an example to demonstrate the use of this element:

XML

```
<icegrid>
    <application name="SampleApp">
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <server id="${id}" activation="manual" .../>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate" id="TheServer"/>
        </node>
    </application>
</icegrid>
```

See Also

- IceGrid Templates
- Server Descriptor Element
- Server-Instance Descriptor Element
- Application Descriptor Element
- Using Descriptor Variables and Parameters

Service Descriptor Element

A `service` element defines an IceBox service. It typically contains at least one `adapter` element, and may supply additional information such as configuration properties.

This element may only appear as a child of an `icebox` element or a `service-template` element.

The following attributes are supported:

Attribute	Description	Required
`entry`	Specifies the entry point of this service.	Yes
`name`	Specifies the name of this service. Within the service, child elements can refer to its name using the reserved variable `${service}`.	Yes

An optional nested `description` element provides free-form descriptive text.

Here is an example to demonstrate the use of this element:

XML

```
<icebox id="MyIceBox" ...>
    <service name="Service1" entry="service1:Create">
        <description>A description of this service.</description>
        <property name="ServiceName" value="${service}"/>
        <adapter name="MyAdapter" id="${service}Adapter" .../>
    </service>
    <service name="Service2" entry="service2:Create"/>
</icebox>
```

See Also

- IceBox
- Adapter Descriptor Element
- Properties Descriptor Element
- IceBox Descriptor Element
- Service-Template Descriptor Element
- IceBox Integration with IceGrid

Service-Instance Descriptor Element

A `service-instance` element creates an instance of a `service-template` element in an IceBox server. It may supply additional information such as configuration properties.

This element may only appear as a child of an `icebox` element.

The following attributes are supported:

Attribute	Description	Required
`template`	Identifies the service template.	Yes

All other attributes of the element must correspond to parameters declared by the template. The `service-instance` element must provide a value for each parameter that does not have a default value supplied by the template.

Here is an example to demonstrate the use of this element:

XML

```
<icebox id="IceBoxServer" ...>
    <service-instance template="ServiceTemplate" name="Service1">
        <properties>
            <property name="Debug" value="1"/>
        </properties>
    </service-instance>
</icebox>
```

See Also

- Service-Template Descriptor Element
- IceBox Descriptor Element
- IceGrid Templates
- IceBox Integration with IceGrid

Service-Template Descriptor Element

A `service-template` element defines a template for a `service` element, simplifying the task of deploying multiple instances of the same service definition. The template should contain a parameterized `service` element that is instantiated using a `service-instance` element.

This element may only appear as a child of an `application` element.

The following attributes are supported:

Attribute	Description	Required
`id`	Specifies the identifier for the service template. This identifier must be unique among all service templates in the application.	Yes

A template may declare parameters that are used to instantiate the `service` element. You can define a default value for each parameter. Parameters without a default value are considered mandatory and values for them must be supplied by the `service-instance` element.

Here is an example to demonstrate the use of this element:

XML

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                <adapter name="${service}" .../>
            </service>
        </service-template>
        <node name="Node1">
            <icebox id="IceBoxServer" ...>
                <service-instance template="ServiceTemplate"
name="Service1"/>
            </icebox>
        </node>
    </application>
</icegrid>
```

See Also

- IceGrid Templates
- Service Descriptor Element
- Service-Instance Descriptor Element
- IceBox Integration with IceGrid
- Using Descriptor Variables and Parameters

Variable Descriptor Element

A `variable` element defines a variable.

This element may only appear as a child of an `application` element or `node` element.

The following attributes are supported:

Attribute	Description	Required
`name`	Specifies the variable name. The value of this variable is substituted whenever its name is used in variable syntax, as in `${name}`.	Yes
`value`	Specifies the variable value. If not defined, the default value is an empty string.	No

Here is an example to demonstrate the use of this element:

XML

```
<icegrid>
    <application name="SampleApp">
        <variable name="Var1" value="foo"/>
        <variable name="Var2" value="${Var1}bar"/>
        ...
    </application>
</icegrid>
```

See Also

- Using Descriptor Variables and Parameters
- Application Descriptor Element
- Node Descriptor Element

Using Command Line Options in Descriptors

Server descriptors and IceBox descriptors may specify command-line options that the node will pass to the program at startup. As the node prepares to execute the server, it assembles the command by appending options to the server executable's pathname.

In XML, you define a command-line option using the `option` element:

XML

```
<server id="Server1" ...>
    <option>--Ice.Trace.Protocol</option>
    ...
</server>
```

The node preserves the order of options, which is especially important for Java servers. For example, JVM options must appear before the class name, as shown below:

XML

```
<server id="JavaServer" exe="java" ...>
    <option>-Xnoclassgc</option>
    <option>ServerClassName</option>
    <option>--Ice.Trace.Protocol</option>
    ...
</server>
```

The node translates these options into the following command:

```
java -Xnoclassgc ServerClassName --Ice.Trace.Protocol
```

See Also

- Server Descriptor Element
- IceBox Descriptor Element

Setting Environment Variables in Descriptors

Server descriptors and IceBox descriptors may specify environment variables that the node will define when starting a server. An environment variable definition uses the familiar *name=value* syntax, and you can also refer to other environment variables within the value. The exact syntax for variable references depends on the platform on which the server's descriptor is deployed.

On a Unix platform, the Bourne shell syntax is required:

```
LD_LIBRARY_PATH=/opt/Ice/lib:$LD_LIBRARY_PATH
```

On a Windows platform, the syntax uses the conventional style:

```
PATH=C:\Ice\lib;%PATH%
```

In XML, the `env` element supplies a definition for an environment variable:

XML

```
<node name="UnixBox">
    <server id="UnixServer" exe="/opt/app/bin/server" ...>
        <env>LD_LIBRARY_PATH=/opt/Ice/lib:$LD_LIBRARY_PATH</env>
        ...
    </server>
</node>
<node name="WindowsBox">
    <server id="WindowsServer" exe="C:/app/bin/server.exe" ...>
        <env>PATH=C:\Ice\lib;%PATH%</env>
        ...
    </server>
</node>
```

If a value refers to an environment variable that is not defined, the reference is substituted with an empty string.

Environment variable definitions may also refer to descriptor variables and template parameters:

XML

```
<node name="UnixBox">
    <server id="UnixServer" exe="/opt/app/bin/server" ...>
        <env>PATH=${server.distrib}/bin:$PATH</env>
        ...
    </server>
</node>
```

On Unix, an environment variable `VAR` can be referenced as `$VAR` or `${VAR}`. You must be careful when using the latter syntax because

IceGrid assumes `${VAR}` refers to a descriptor variable or parameter and will report an error if no match is found. If you prefer to use this style to refer to environment variables, you must escape these occurrences as shown in the example below:

```
<node name="UnixBox">
    <server id="UnixServer" exe="/opt/app/bin/server" ...>
        <env>PATH=${server.distrib}/bin:$${PATH}</env>
        ...
    </server>
</node>
```

IceGrid does not attempt to perform substitution on `$${PATH}`, but rather removes the leading $ character and then performs environment variable substitution on `$${PATH}`.

See Also

- Server Descriptor Element
- IceBox Descriptor Element
- Using Descriptor Variables and Parameters

Using Descriptor Variables and Parameters

Variable descriptors allow you to define commonly-used information once and refer to them symbolically throughout your application descriptors.

On this page:

Descriptor Substitution Syntax

Substitution for a variable or parameter *VP* is attempted whenever the symbol ${*VP*} is encountered, subject to the limitations and rules described below. Substitution is case-sensitive, and a fatal error occurs if *VP* is not defined.

Limitations

Substitution is only performed in string values, and excludes the following cases:

- Identifier of a template descriptor definition

```
<server-template id="${invalid}" ...>
```

- Name of a variable definition

```
<variable name="${invalid}" ...>
```

- Name of a template parameter definition

```
<parameter name="${invalid}" ...>
```

- Name of a template parameter assignment

```
<server-instance template="T" ${invalid}="val" ...>
```

- Name of a node definition

```
<node name="${invalid}" ...>
```

- Name of an application definition

```
<application name="${invalid}" ...>
```

Substitution is not supported for values of other types. The example below demonstrates an invalid use of substitution:

```
<variable name="register" value="true"/>
<node name="Node">
    <server id="Server1" ...>
        <adapter name="Adapter1" register-process=${register} .../>
```

In this case, a variable cannot supply the value of `register-process` because that attribute expects a boolean value, not a string.

Most values are strings, however, so this limitation is rarely a problem.

Escaping a Variable

You can prevent substitution by escaping a variable reference with an additional leading $ character. For example, in order to assign the literal string `${abc}` to a variable, you must escape it as shown below:

```
<variable name="x" value="$${abc}"/>
```

The extra $ symbol is only meaningful when immediately preceding a variable reference, therefore text such as `US$$55` is not modified. Each occurrence of the characters `$$` preceding a variable reference is replaced with a single $ character, and that character does not initiate a variable reference. Consider these examples:

```
<variable name="a" value="hi"/>
<variable name="b" value="$${a}"/>
<variable name="c" value="$$${a}"/>
<variable name="d" value="$$$${a}"/>
```

After substitution, b has the value `${a}`, c has the value `$hi`, and d has the value `$${a}`.

Special Descriptor Variables

IceGrid defines a set of read-only variables to hold information that may be of use to descriptors. The names of these variables are reserved and cannot be used as variable or parameter names. The table describes the purpose of each variable and defines the context in which it is valid.

Reserved Name	Description
`application`	The name of the enclosing application.
`application.distrib`	The pathname of the enclosing application's distribution directory, and an alias for `${node.datadir}/distrib/${application}`.
`node`	The name of the enclosing node.
`node.os`	The name of the enclosing node's operating system. On Unix, this is value is provided by `uname`. On Windows, the value is `Windows`.
`node.hostname`	The host name of the enclosing node.
`node.release`	The operating system release of the enclosing node. On Unix, this value is provided by `uname`. On Windows, the value is obtained from the `OSVERSIONINFO` data structure.
`node.version`	The operating system version of the enclosing node. On Unix, this value is provided by `uname`. On Windows, the value represents the current service pack level.

`node.machine`	The machine hardware name of the enclosing node. On Unix, this value is provided by `uname`. On Windows, the value can be x86, x64, or IA64, depending on the machine architecture.
`node.data`	The absolute pathname of the enclosing node's data directory.
`server`	The ID of the enclosing server.
`server.distrib`	The pathname of the enclosing server's distribution directory, and an alias for `${node.datadir}/servers/${server}/distrib`.
`server.data`	The pathname of the enclosing server's user data directory, and an alias for `${node.data}/servers/${server}/data`.
`service`	The name of the enclosing service.
`service.data`	The pathname of the enclosing service's user data directory, and an alias for `${node.data}/servers/${server}/data_${service}`.
`session.id`	The client session identifier. For sessions created with a user name and password, the value is the user ID; for sessions created from a secure connection, the value is the distinguished name associated with the connection.

The availability of a variable is easily determined in some cases, but may not be readily apparent in others. For example, the following example represents a valid use of the `${node}` variable:

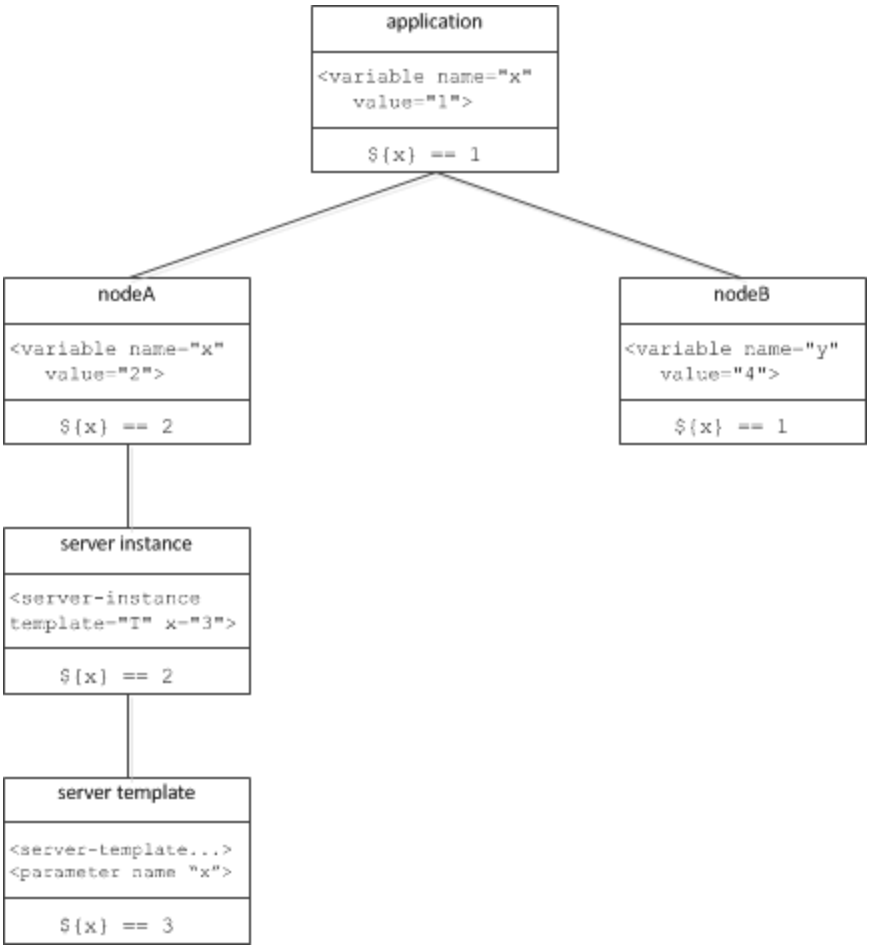
XML

```
<icegrid>
    <application name="App">
        <server-template id="T" ...>
            <parameter name="id"/>
            <server id="${id}" ...>
                <property name="NodeName" value="${node}"/>
                ...
            </server>
        </server-template>
        <node name="TheNode">
            <server-instance template="T" id="TheServer"/>
        </node>
    </application>
</icegrid>
```

Although the server template descriptor is defined as a child of an application descriptor, its variables are not evaluated until it is instantiated. Since a template *instance* is always enclosed within a node, it is able to use the `${node}` variable.

Descriptor Variable Scoping Rules

Descriptors may only define variables at the application and node levels. Each node introduces a new scope, such that defining a variable at the node level overrides (but does not modify) the value of an application variable with the same name. Similarly, a template parameter overrides the value of a variable with the same name in an enclosing scope. A descriptor may refer to a variable defined in any enclosing scope, but its value is determined by the nearest scope. The following figure illustrates these concepts:



Variable scoping semantics.

In this diagram, the variable x is defined at the application level with the value 1. In `nodeA`, x is overridden with the value 2, whereas x remains unchanged in `nodeB`. Within the context of `nodeA`, x continues to have the value 2 in a server instance definition. However, when x is used as the name of a template parameter, the node's definition of x is overridden and x has the value 3 in the template's scope.

Resolving a Reference

To resolve a variable reference `${var}`, IceGrid searches for a definition of `var` using the following order of precedence:

1. Pre-defined variables
2. Template parameters, if applicable
3. Node variables, if applicable
4. Application variables

After the initial substitution, any remaining references are resolved recursively using the following order of precedence:

1. Pre-defined variables
2. Node variables, if applicable
3. Application variables

Template Parameters

Template parameters are not visible in nested template instances. This situation can only occur when an IceBox server template instantiates a service template, as shown in the following example:

XML

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                ...
                <property name="${name}.Identity"
value="${id}-${name}"/> <!-- WRONG! -->
            </service>
        </service-template>
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <icebox id="${id}" endpoints="default" ...>
                <service-instance template="ServiceTemplate"
name="Service1"/>
            </icebox>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate"
id="IceBoxServer"/>
        </node>
    </application>
</icegrid>
```

The service template incorrectly refers to `id`, which is a parameter of the server template.

Template parameters can be referenced only in the body of a template; they cannot be used to define other parameters. For example, the following is illegal:

XML

```
<server-template id="ServerTemplate">
    <parameter name="par1"/>
    <parameter name="par2" default="${par1}"/>
    ...
</server-template>
```

Modifying a Variable

A variable definition can be overridden in an inner scope, but the inner definition does not modify the outer variable. If a variable is defined multiple times in the same scope (which is only relevant in XML definitions), the most recent definition is used for all references to that variable. Consider the following example:

XML

```
<application name="MyApp">
    <variable name="x" value="1"/>
    <variable name="y" value="${x}"/>
    <variable name="x" value="2"/>
    ...
</application>
```

When descriptors such as these are created, IceGrid validates their variable references but does not perform substitution until the descriptor is acted upon (such as when a node is generating a configuration file for a server). As a result, the value of y in the above example is 2 beca use that is the most recent definition of x.

See Also

- Variable Descriptor Element
- IceGrid Templates
- Application Distribution
- Variables in IceGrid Descriptors

IceGrid Property Set Semantics

Ice servers and clients are configured with properties. For servers deployed with IceGrid, these properties are automatically generated into a configuration file from the information contained in the application descriptor. The settings in that configuration file are passed to server via the `--Ice.Config` command-line option.

Property descriptors allow you to define property sets to efficiently manage and specify properties. Here are some of the benefits of using property sets:

- You can define sets of properties at the application or node element level and reference these properties in other property sets.
- You can specify properties for a specific server or service instance.

There are two kinds of property sets:

- Named property sets
  Named property sets are defined at the application or node level. They are useful only as the target of references from other property sets. Specifically, a named property set has no effect unless you reference it from a server descriptor.
- Unnamed property sets
  Unnamed property sets can be defined in server, service, icebox, server-instance or service-instance elements and define the properties for a server or service. Unnamed property sets can reference named property sets.

Named and unnamed property sets are defined with the same properties descriptor. The context and the attributes of a `properties` element distinguish named property sets from unnamed property sets. Here is an example that defines a named and an unnamed property set:

XML

```
<application name="App">
    <properties id="Debug">
        <property name="UseDebug" value="1"/>
    </properties>

    <node name="TheNode">
        <server id="TheServer" exe="./server">
            <properties>
                <property name="Identity" value="hello"/>
            </properties>
        </server>
    </node>
</application>
```

In this example, we define the named property set `Debug` and the unnamed property set of the server `TheServer`. The server configuration will contain only the `Identity` property because the server property set does not reference the `Debug` named property set.

The `properties` element is used to reference a named property set: if a `properties` element appears inside another `properties` element, it is a reference to another property set and it must specify the `refid` attribute. With the previous example, to reference the `Debug` property set, we would write the following:

XML

```
<application name="App">
    <properties id="Debug">
        <property name="UseDebug" value="1"/>
    </properties>

    <node name="TheNode">
        <server id="TheServer" exe="./server">
            <properties>
                <properties refid="Debug"/>
                <property name="Identity" value="hello"/>
        </properties>
        </server>
    </node>
</application>
```

Property sets, whether named or unnamed, are evaluated as follows:

1. Within a `properties` element, IceGrid locates all references to named property sets and evaluates all property settings in the referenced property sets.
2. Explicit property definitions following any named references are then evaluated and added to the property set formed in the preceding step. This means that explicit property settings override corresponding settings in any referenced property sets.

It is illegal to define a reference to a property set after setting a property value, so references to property sets must precede property definitions. For example, the following is illegal:

XML

```
<properties>
    <property name="Prop1" value="Value1"/>
    <properties refid="Ref1"/>
</properties>
```

Just as the order of the property definitions is important, the order of property set references is also important. For example, the following two property sets are not equivalent:

XML

```
<properties>
    <properties refid="Ref1"/>
    <properties refid="Ref2"/>
</properties>

<properties>
    <properties refid="Ref2"/>
    <properties refid="Ref1"/>
</properties>
```

Named property sets are evaluated at the point of definition. If you reference other property sets or use variables in a named property set definition, you must make sure that the referenced property sets or variables are defined in the same scope. For example, the following is correct:

XML

```
<application name="App">

    <variable name="level" value="1"/>

    <properties id="DebugApp">
        <property name="DebugLevel value="${level}">
    </properties>

</application>
```

However, the following example is wrong because the `${level}` variable is not defined at the `application` scope:

XML

```
<application name="App">

    <properties id="DebugApp">
        <property name="DebugLevel value="${level}">
    </properties>

    <node name="TheNode">
        <variable name="level" value="1"/>
    </node>

</application>
```

If both the `application` and the `node` define the `${level}` variable, the value of the `${level}` variable in the `DebugApp` property set

will be the value of the variable defined in the application descriptor.

So far, we have seen the definition of an unnamed property set only in a server descriptor. However, it is also possible to define an unnamed property set for server or service instances. This is a good way to specify or override properties specific to a server or service instance. For example:

XML

```
<application name="TheApp">
  <server-template id="Template">

    <parameter name="instance-name"/>

    <server id="${instance-name}" exe="./server">
      <properties>
        <property name="Timeout" value="30"/>
      </properties>
    </server>
  </server-template>

  <node name="TheNode">
    <server-instance template="Template" instance-name="MyInst">
      <properties>
        <property name="Debug" value="1"/>
        <property name="Timeout" value="-1"/>
      </properties>
    </server-instance>
  </node>
</application>
```

Here, the server instance overrides the `Timeout` property and defines an additional `Debug` property.

The server or service instance properties are evaluated as follows:

1. The unnamed property set from the template server or service descriptor is evaluated.
2. The unnamed property set from the server or service instance descriptor is evaluated and the resulting properties are added to the property set formed in the preceding step. This means that property settings in a server or service instance descriptor override corresponding settings in a template server or service descriptor.

The server or service instance unnamed property set and its parameters provide two different ways to customize the properties of a server or service template instance. It might not always be obvious which method to use: is it better to use a parameter to parameterize a given property or is it better to just specify it in the server or service instance property set?

For example, in the previous descriptor, we could have used a parameter with a default value for the `Timeout` property:

XML

```
<application name="TheApp">
  <server-template id="Template">

    <parameter name="instance-name"/>
    <parameter name="timeout" default="30"/>

    <server id="${instance-name}" exe="./server">
      <properties>
        <property name="Timeout" value="${timeout}"/>
      </properties>
    </server>
  </server-template>

  <node name="TheNode">
    <server-instance template="Template"
instance-name="MyInst" timeout="-1">
      <properties>
        <property name="Debug" value="1"/>
      </properties>
    </server-instance>
  </node>
</application>
```

Here are some guidelines to help you decide whether to use a parameter or a property:

- Use a parameter for a property that should always be set.
- Use a parameter if you want to make the property obvious to the reader and user of the template.
- Do not use a parameter for optional properties if you want to rely on a default value for the server.
- Do not use parameters for properties that are rarely used.

See Also

- Properties and Configuration
- Using IceGrid Deployment
- Properties Descriptor Element
- Application Descriptor Element
- Node Descriptor Element
- Server-Instance Descriptor Element
- Service-Instance Descriptor Element

IceGrid XML Features

IceGrid provides some convenient features to simplify the task of defining descriptors in XML.

On this page:

Adding Flexibility with Targets

An IceGrid XML file may contain optional definitions that are deployed only when specifically requested. These definitions are called targets and must be defined within a `target` element. The elements that may legally appear within a `target` element are determined by its enclosing element. For example, a `node` element is legal inside a `target` element of an `application` element, but not inside a `target` el ement of a `server` element. Each `target` element must define a value for the `name` attribute, but names are not required to be unique. Rather, targets should be considered as optional components or features of an application that are deployed in certain circumstances.

The example below defines targets named `debug` that, if requested during deployment, configure their servers with an additional property:

XML

```
<icegrid>
    <application name="MyApp">
        <node name="Node">
            <server id="Server1" ...>
                <target name="debug">
                    <property name="Ice.Trace.Network" value="2"/>
                </target>
                ...
            </server>
            <server id="Server2" ...>
                <target name="debug">
                    <property name="Ice.Trace.Network" value="2"/>
                </target>
                ...
            </server>
        </node>
    </application>
</icegrid>
```

Target names specified in an `icegridadmin` command can be unqualified names like `debug`, in which case every target with that name is deployed, regardless of the target's nesting level. If you want to deploy targets more selectively, you can specify a fully-qualified name instead. A fully-qualified target name consists of its unqualified name prefaced by the names or identifiers of each enclosing element. For instance, a fully-qualified target name from the example above is `MyApp.Node.Server1.debug`.

Including Descriptor Files

You can include the contents of another XML file into the current file using the `include` element, which is replaced with the contents of the included file. The elements in the included file must be enclosed in an `icegrid` element, as shown in the following example:

XML

```
<!-- File: A.xml -->
<icegrid>
    <server-template id="ServerTemplate">
        <parameter name="id"/>
        ...
    </server-template>
</icegrid>

<!-- File: B.xml -->
<icegrid>
    <application name="MyApp">
        <include file="A.xml"/>
        <node name="Node">
            <server-instance template="ServerTemplate" .../>
        </node>
    </application>
</icegrid>
```

In `B.xml`, the `include` element identifies the name of the file to include using the `file` attribute. The top-level `icegrid` element is discarded from `A.xml` and its contents are inserted at the position of the `include` element in `B.xml`.

Note that the file name of an included file is relative to the application descriptor, not relative to the working directory.

You can include specific targets from a file by specifying their names in the optional `targets` attribute. If multiple targets are included, their names must be separated by whitespace. The example below illustrates the use of a target:

XML

```
<!-- File: A.xml -->
<icegrid>
    <server-template id="ServerTemplate">
        <parameter name="id"/>
        ...
    </server-template>
    <target name="targetA">
        <server-template id="AnotherTemplate">
            ...
        </server-template>
    </target>
</icegrid>

<!-- File: B.xml -->
<icegrid>
    <application name="MyApp">
        <include file="A.xml" targets="targetA"/>
        <node name="Node">
            <server-instance template="ServerTemplate" .../>
            <server-instance template="AnotherTemplate" .../>
        </node>
    </application>
</icegrid>
```

See Also

- Using IceGrid Deployment
- icegridadmin Command Line Tool

IceGrid Server Reference

Topics

icegridregistry

The IceGrid registry is a centralized repository of information, including deployed applications and well-known objects. A registry can optionally be collocated with an IceGrid node, which conserves resources and can be convenient during development and testing. The registry server is implemented by the `icegridregistry` executable.

On this page:

Command Line Options for `icegridregistry`

The registry supports the following command-line options:

```
$ icegridregistry -h
Usage: icegridregistry [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
--nowarn             Don't print any security warnings.
--readonly           Start the master registry in read-only mode.
--initdb-from-replica <replica>
                     Initialize the database from the given replica.
```

The `--readonly` option prevents any updates to the registry's database; it also prevents slaves from synchronizing their databases with this master. This option is useful when you need to verify that the master registry's database is correct after promoting a slave to become the new master. The `--initdb-from-replica` option, added in Ice 3.5.1, allows you to initialize the database from another registry replica. This option is useful when you need to start a new master with the contents of a slave database.

Additional command line options are supported, including those that allow the registry to run as a Windows service or Unix daemon, and Ice includes a utility to help you install an IceGrid registry as a Windows service.

Configuring Registry Endpoints

The IceGrid registry creates up to five sets of endpoints, configured with the following properties:

- `IceGrid.Registry.Client.Endpoints`
  Client-side endpoints supporting the following interfaces:
  - `Ice::Locator`
  - `IceGrid::Query`
  - `IceGrid::Registry`
  - `IceGrid::Session`
  - `IceGrid::AdminSession`
  - `IceGrid::Admin`

  > There are security implications in allowing access to administrative sessions, as explained in the next section.

- `IceGrid.Registry.Server.Endpoints`
  Server-side endpoints for object adapter registration.

- `IceGrid.Registry.SessionManager.Endpoints`
  Optional endpoints for delegating session authentication to a Glacier2 router.

- `IceGrid.Registry.AdminSessionManager.Endpoints`
  Optional endpoints for delegating administrative session authentication to a Glacier2 router.
- `IceGrid.Registry.Internal.Endpoints`
  Internal endpoints used by IceGrid nodes and registry replicas. This property must be defined even if no nodes or replicas are being used.

Registry Security Considerations

A client that successfully establishes an administrative session with the registry has the ability to compromise the security of the registry host. As a result, it is imperative that you configure the registry carefully if you intend to allow the use of administrative sessions.

Administrative sessions are disabled unless you explicitly configure the registry to use an authentication mechanism. To allow authentication with a user name and password, you can specify a password file using the property `IceGrid.Registry.AdminCryptPasswords` or use your own permissions verifier object by supplying its proxy in the property `IceGrid.Registry.AdminPermissionsVerifier`. Your verifier object must implement the `Glacier2::PermissionsVerifier` interface.

To authenticate administrative clients using their SSL connections, define `IceGrid.Registry.AdminSSLPermissionsVerifier` with the proxy of a verifier object that implements the `Glacier2::SSLPermissionsVerifier` interface.

Configuring the Registry's Database Directory

You must provide an empty directory in which the registry can initialize its database. The path name of this directory is supplied by the configuration property `IceGrid.Registry.LMDB.Path`.

The files in this directory must not be edited manually, but rather indirectly using one of the administrative tools. To clear a registry's database, first ensure the server is not currently running, then remove all of the files in its data directory and restart the server.

Registry Configuration Example

The registry requires values for the three mandatory endpoint properties, as well as the database directory property, as shown in the following example:

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.LMDB.Path=/opt/ripper/registry
```

In addition, we also recommend defining `IceGrid.InstanceName`, whose value affects the identities of the registry's well-known objects.

The remaining configuration properties are discussed in IceGrid.*.

See Also

- Using IceGrid Deployment
- Well-Known Objects
- Promoting a Registry Slave
- Windows Services
- Resource Allocation using IceGrid Sessions
- IceGrid Administrative Sessions
- Glacier2 Integration with IceGrid
- icegridadmin Command Line Tool
- Securing a Glacier2 Router
- IceGrid Persistent Data
- Well-Known Registry Objects
- IceGrid.*

icegridnode

An IceGrid node is a process that activates, monitors, and deactivates registered server processes. You can run any number of nodes in a domain, but typically there is one node per host. A node must be running on each host on which servers are activated automatically, and nodes cannot run without an IceGrid registry.

The IceGrid node server is implemented by the `icegridnode` executable. If you wish to run a registry and node in one process, `icegridnode` is the executable you must use.

On this page:

Command Line Options for `icegridnode`

The node supports the following command-line options:

```
Usage: icegridnode [options]
Options:
-h, --help            Show this message.
-v, --version         Display the Ice version.
--nowarn              Don't print any security warnings.
--readonly            Start the collocated master registry in
                      read-only mode.
--initdb-from-replica <replica>
                      Initialize the collocated registry database from
the
                      given replica.

--deploy DESCRIPTOR [TARGET1 [TARGET2 ...]]
                      Add or update descriptor in file DESCRIPTOR,
                      with optional targets.
```

If you are running the node with a collocated registry, the `--readonly` option prevents any updates to the registry's database; it also prevents slaves from synchronizing their databases with this master. This option is useful when you need to verify that the master registry's database is correct after promoting a slave to become the new master. The `--initdb-from-replica` option, added in Ice 3.5.1, allows you to initialize the database from another registry replica. This option is useful when you need to start a new master with the contents of a slave database.

The `--deploy` option allows an application to be deployed automatically as the node process starts, which can be especially useful during testing. The command expects the name of the XML deployment file, and optionally allows the names of the individual targets within the file to be specified.

Additional command line options are supported, including those that allow the node to run as a Windows service or Unix daemon, and Ice includes a utility to help you install an IceGrid node as a Windows service.

Configuring Node Endpoints

The IceGrid node's endpoints are defined by the `IceGrid.Node.Endpoints` property and must be accessible to the registry. It is not necessary to use a fixed port because each node contacts the registry at startup to provide its current endpoint information.

Node Security Considerations

It is important that you give careful consideration to the permissions of the account under which the node runs. If the servers that the node will activate have no special access requirements, and all of the servers can use the same account, it is recommended that you do not run the node under an account with system privileges, such as the root account on Unix or the Administrator account on Windows.

Configuring a Data Directory for the Node

The node requires an empty directory that it can use to store server files. The pathname of this directory is supplied by the configuration property `IceGrid.Node.Data`. To clear a node's state, first ensure the server is not currently running, then remove all of the files in its data directory and restart the server.

> The node's data directory may also contain files and subdirectories used by your application's servers, such as configuration files and Freeze database environments. Before destroying the contents of the node's data directory, make sure that all servers are stopped and any important files are backed up.

When running a collocated node and registry server, we recommend using separate directories for the registry and node data directories.

Node Configuration Example

A minimal node configuration is shown in the following example:

```
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node1
IceGrid.Node.Data=/opt/ripper/node

Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

The value of the `IceGrid.Node.Name` property must match that of a deployed node known by the registry.

The `Ice.Default.Locator` property is used by the node to contact the registry. The value is a proxy that contains the registry's client endpoints. (You can avoid the need to define `Ice.Default.Locator` by using IceLocatorDiscovery.)

If you wish to run a collocated registry and node server, enable the property `IceGrid.Node.CollocateRegistry` and include the registry's configuration properties.

The remaining configuration properties are discussed in IceGrid.*.

See Also

- IceGrid Server Activation
- Promoting a Registry Slave
- IceGrid Persistent Data
- Getting Started with IceGrid
- Using IceGrid Deployment
- Windows Services
- IceGrid.*

Well-Known Registry Objects

On this page:

Default Identities of Registry Objects

The IceGrid registry hosts several well-known objects. The following table shows the default identities of these objects and their corresponding Slice interfaces:

Default Identity	Interface
`IceGrid/AdminSessionManager`	`Glacier2::SessionManager`
`IceGrid/AdminSessionManager-`*`replica`*	`Glacier2::SessionManager`
`IceGrid/AdminSSLSessionManager`	`Glacier2::SSLSessionManager`
`IceGrid/AdminSSLSessionManager-`*`replica`*	`Glacier2::SSLSessionManager`
`IceGrid/Locator`	`Ice::Locator`
`IceGrid/Query`	`IceGrid::Query`
`IceGrid/Registry`	`IceGrid::Registry`
`IceGrid/Registry-`*`replica`*	`IceGrid::Registry`
`IceGrid/RegistryUserAccountMapper`	`IceGrid::UserAccountMapper`
`IceGrid/RegistryUserAccountMapper-`*`replica`*	`IceGrid::UserAccountMapper`
`IceGrid/SessionManager`	`Glacier2::SessionManager`
`IceGrid/SSLSessionManager`	`Glacier2::SSLSessionManager`
`Ice/LocatorFinder`	`Ice::LocatorFinder`

You can assign unique identities to these objects by configuring the `IceGrid.InstanceName` property, as shown in the following example:

```
IceGrid.InstanceName=MP3Grid
```

This property changes the identities of the well-known objects to use `MP3Grid` instead of `IceGrid` as the identity category. For example, the identity of the locator becomes `MP3Grid/Locator`.

> Changes to `IceGrid.InstanceName` do not affect the identity of the `Ice/LocatorFinder` object. See the next section for more information.

The client's configuration must also be changed to reflect the new identity:

```
Ice.Default.Locator=MP3Grid/Locator:tcp -h registryhost -p 4061
```

Furthermore, any uses of these identities in application code must be updated as well.

Using the `LocatorFinder` Interface

Ice requires all locator implementations to support the `Ice::LocatorFinder` interface:

Slice

```
module Ice
{
    interface LocatorFinder
    {
        Locator* getLocator();
    }
}
```

An object supporting this interface must be available with the identity `Ice/LocatorFinder`. By knowing the host and port of a locator's client endpoints, a client can discover the locator's proxy at run time with a call to `getLocator`:

C++11 C++98

```
auto prx = communicator->stringToProxy("Ice/LocatorFinder:tcp -p 4061 -h
gridhost");
auto finder = Ice::checkedCast<Ice::LocatorFinderPrx>(prx);
auto locator = finder->getLocator();
communicator->setDefaultLocator(locator);
```

```
Ice::ObjectPrx prx = communicator->stringToProxy("Ice/LocatorFinder:tcp
-p 4061 -h gridhost");
Ice::LocatorFinderPrx finder = Ice::LocatorFinderPrx::checkedCast(prx);
Ice::LocatorPrx locator = finder->getLocator();
communicator->setDefaultLocator(locator);
```

See Also

- Well-Known Objects

IceGrid Persistent Data

The IceGrid registry and node both store information in files. This section describes what type of information the registry and node are storing, and discusses backup and recovery techniques.

On this page:

Registry Database

Data Stored

The registry stores the following data in the LMDB database specified through the IceGrid.Registry.LMDB.Path property:

- Applications deployed using the `addApplication` operation on the `IceGrid::Admin` interface (which includes the IceGrid GUI and command-line administrative clients). Applications describe servers, well-known objects, object adapters, replica groups, and allocatable objects. Applications can be removed with the `removeApplication` operation.
- Well-known objects registered using the `addObject` and `addObjectWithType` operations on the `IceGrid::Admin` interface. Well-known objects added by these operations can be removed using the `removeObject` operation.
- Adapter endpoints registered dynamically by servers using the `Ice::LocatorRegistry` interface. The property `IceGrid.Registry.DynamicRegistration` must be set to a value larger than zero to allow the dynamic registration of object adapters. These adapters can be removed using the `removeAdapter` operation.
- Some internal proxies used by the registry to contact nodes and other registry replicas during startup. The proxies enable the registry to notify these entities about the registry's availability.

Client session and administrative sessions established with the IceGrid registry are not stored in this database. If the registry is restarted, these sessions must be recreated. For client sessions in particular, this implies that objects allocated using the allocation mechanism will no longer be allocated once the IceGrid registry restarts.

If the registry's database is corrupted or lost, you must recover the deployed applications, the well-known objects, and the adapter endpoints. You do not need to worry about the internal proxies stored by the registry, as the nodes and registry replicas will eventually contact the registry again.

Depending on your deployed applications and your use of the registry, you should consider backing up the registry's database, especially if you cannot easily recover the persistent information.

For example, if you rely on dynamically-registered adapters, or on well-known objects registered programmatically via the `IceGrid::Admin` interface, you should back up the registry database because recovering this information may be difficult. On the other hand, if you only deploy a few applications from XML files, you can easily recover the applications by redeploying their XML files, and therefore backing up the database may be unnecessary.

Be aware that restarting the registry with an empty database may cause the server information stored by the nodes to be deleted. This can be an issue if the deployed servers have databases stored in the node data directory. The Node Persistent Data section below provides more information on this subject.

Limits Imposed by the Registry Database

Key Size

A LMDB database consists of one or more persistent key-value maps, and the size of the keys in these maps is limited to 511 bytes. For example, IceGrid stores applications in a persistent map where the keys are the application names, encoded using the Ice encoding for strings, and this LMDB limitation means you cannot select an application name with an arbitrary size.

IceGrid Registry Database Key	Slice Type	Max Encoded Size
Adapter ID	`string`	511 bytes
Application Name	`string`	511 bytes

Well-known Object Identity	`Ice::Identity`	511 bytes
Well-known Object Type	`string`	511 bytes

If you attempt to save an application, adapter ID or well-known object that is too large for the LMDB database, IceGrid will throw an `IceGrid::DeploymentException` or an `Ice::UnknownException` depending on the operation invoked.

> This maximum key size is not configurable. If you exceed this limit, you need to shorten the corresponding name, ID or identity.

Map Size

A LMDB database has a maximum size, known as its map size. The IceGrid registry database can store up to `IceGrid.Registry.LMDB.MapSize` megabytes of data in its database; any attempt to store more data will fail with an `Ice::UnknownException`. If you exceed this limit, increase `IceGrid.Registry.LMDB.MapSize` and restart the IceGrid registry.

If you don't set `IceGrid.Registry.LMDB.MapSize`, or set it to 0, IceGrid uses a map size of 10 MB on Windows, and 100 MB on Linux and OS X.

On Windows, LMDB immediately allocates a file with the given map size, while on Linux and OS X LMDB uses sparse files and the allocated data file starts small and grows as needed, until it reaches the configured limit.

> The default `MapSize` provided by the IceGrid registry is expected to be sufficient for most applications. Unless you have an extremely large IceGrid deployment on Windows, we recommend keeping the default setting.
>
> Use the `mdb_stat` utility to monitor the pages used by your IceGrid registry database. The example below shows a LMDB database with the default size on Linux (100 MB):

```
$ mdb_stat -e registry
Environment Info
  Map address: (nil)
  Map size: 104857600
  Page size: 4096
  Max pages: 25600
  Number of pages used: 18
  Last transaction ID: 8
  Max readers: 126
  Number of readers used: 0
Status of Main DB
  Tree depth: 1
  Branch pages: 0
  Leaf pages: 1
  Overflow pages: 0
  Entries: 8
```

Backing up the Registry Database

You should consider making regular backups of your IceGrid registry database. We recommend using one of the following tools to perform backups while the IceGrid registry is running:

- `icegriddb` with the `--export` option
- `mdb_copy` or `mdb_dump`

Node Persistent Data

Each IceGrid node stores information in a directory specified through its IceGrid.Node.Data property - the node's data directory. In IceGrid descriptors, the `node.data` variable is substituted with the path of the node data directory.

Each node stores configuration files, user data and occasionally database environments and distribution data, for each server in a sub-directory named `servers/<server ID>`, where `<server ID>` represents the unique server ID of that server. Per-server or per-service user data can be stored in the sub-directories named `servers/<server ID>/data` and `servers/<server ID>/data_<service name>`. In IceGrid descriptors, the `server.data` and `service.data` variables are replaced with the paths of the server or service data directory.

The node's data directory also contains a `distrib` directory to store per-application distribution data. This directory contains a subdirectory for each application that specifies a distribution and has a server deployed on the node.

If a server directory is deleted, the node recreates it at startup. The node will also recreate the server configuration files and the database environment directories. However, the node cannot restore the prior contents of a server's database environment or user data directory. It is your responsibility to back up the user data and database environment directories and restore them when necessary. If the server or application distribution data is deleted from the node's data directory, you can easily recover the deleted information by patching these distributions again using the IceGrid administrative tools.

If you store your server database environments outside the node's data directory (such as in a directory that is regularly backed up), or if you do not have any database environments inside the node's data directory, you do not need to back up the contents of the node's data directory.

See Also

- Using IceGrid Deployment
- icegridadmin Command Line Tool
- Well-Known Objects
- Object Adapter Endpoints
- Resource Allocation using IceGrid Sessions
- IceGrid Administrative Sessions
- Application Distribution
- IceGrid.*

Promoting a Registry Slave

In a replicated IceGrid deployment, you may need to promote a slave to be the new master if the current master becomes unavailable. For example, this situation can occur when the original master cannot be restarted immediately due to a hardware problem, or when your application requires a feature that is only accessible via the master, such as the resource allocation facility or the ability to modify the deployment data.

To promote a slave to become the new master, you must shut down the slave and change its `IceGrid.Registry.ReplicaName` property to `Master` (or remove the property altogether). On restart, the new master notifies the nodes and registries that were active before it was shut down. An inactive registry or node will eventually connect to the new master if its default locator proxy contains the endpoint of the new master registry or the endpoint of a slave that is connected to the new master. If you cannot afford any down-time of the registry and want to minimize the down-time of the master, you should run at least two slaves. That way, if the master becomes unavailable, there will always be one registry available while you promote one of the slaves.

With Ice versions 3.5.0 and prior, a slave synchronizes its database upon connecting to the new master, therefore it is imperative that you promote a slave whose database is valid and up-to-date. To verify that the promoted master database is up-to-date, you can start the new master with the `--readonly` command-line option. While this option is in force, the new master does not update its database, and slaves do not synchronize their databases. You can review the master database with the IceGrid administrative tools and, if the deployment looks correct, you can restart the master without the `--readonly` option to permit updates and slave synchronization.

Starting with Ice 3.5.1, a check has been added to ensure that an out-of-date master database cannot overwrite a newer slave database. In such a scenario, an error message is printed out on the consoles of the IceGrid master and slave, and the slave connection to the master is aborted. You can still connect to the slave and master with the IceGrid administrative tools to verify their databases. Once you determine which database to keep, you can restart either the slave or the master with the `--initdb-from-replica=<name>` option to initialize the out-of-date database with the correct version. This check only works when both the master and slave use Ice 3.5.1 or later.

Note that there is nothing to prevent you from running two masters. If you start two masters and they contain different versions of the deployment information, some slaves and nodes might get updated with out-of-date deployment information (causing some of your servers to be deactivated). You can correct the problem by shutting down the faulty master, but it is important to keep this issue in mind when you restart a master since it might disrupt your applications.

See Also

- Registry Replication
- Resource Allocation using IceGrid Sessions
- Using IceGrid Deployment
- icegridadmin Command Line Tool

IceGrid and the Administrative Facility

The Ice administrative facility provides a general purpose solution for administering individual Ice programs. IceGrid extends this functionality in several convenient ways:

- IceGrid automatically enables the facility in deployed servers, in its nodes and in its registry replicas.
- IceGrid uses the `Process` facet to terminate an active server, giving it an opportunity to perform an orderly shutdown.
- IceGrid provides a secure mechanism for invoking administrative operations on deployed servers, IceGrid nodes and IceGrid registry replicas.

The IceGrid administrative tools in turn use IceGrid's extended administrative facility to:

- display the properties of servers and services.
- attach remote loggers to the Logger of servers, services, nodes and registry replicas.
- manipulate and monitor IceBox services.

We discuss each of these items in separate sections below.

On this page:

Enabling the Administrative Facility for a Deployed Server, Node or Registry Replica

As we saw in our deployment example, the configuration properties for a deployed server include definitions for the following properties:

- `Ice.Admin.Endpoints`
- `Ice.Admin.ServerId`

The definition of `Ice.Admin.Endpoints` enables the Administrative Facility.

Ice.Admin.Endpoints for Servers

If a server's descriptor does not set `Ice.Admin.Enabled` and does not supply a value for `Ice.Admin.Endpoints`, IceGrid supplies a default value for `Ice.Admin.Endpoints` as shown below:

```
Ice.Admin.Endpoints=tcp -h 127.0.0.1
```

For security reasons, IceGrid specifies the local host interface (`127.0.0.1`) so that administrative access is limited to clients running on the same host. This configuration permits the IceGrid node to invoke operations on the server's `admin` object, but prevents remote access unless the client establishes an IceGrid administrative session.

Specifying a fixed port is unnecessary because the server registers its endpoints with IceGrid upon each new activation.

admin Objects in Nodes and Registry Replicas

Each IceGrid node and IceGrid registry replica provides by default an admin object hosted in the `IceGrid.Node` object adapter (for nodes) or in the `IceGrid.Registry.Internal` object adapter (for registry replicas). If you don't want to an admin object in a node or registry replica, set Ice.Admin.Enabled to 0 or a negative value in this node or registry's configuration.

Each of these admin objects carries all the built-in facets, currently `Logger`, Metrics, Process and Properties. Proxies to these admin objects

can be retrieved through the `getNodeAdmin` and `getRegistryAdmin` operations described below.

Deactivating a Deployed Server

An IceGrid node uses the `Ice::Process` interface to gracefully deactivate a server. In programs using Ice 3.3 or later, this interface is implemented by the administrative facet named `Process`. In earlier versions of Ice, an object adapter implemented this interface in a special servant if the adapter's `RegisterProcess` property was enabled.

Regardless of version, the Ice run time registers an `Ice::Process` proxy with the IceGrid registry when properly configured. Registration normally occurs during communicator initialization, but it can be delayed when a server needs to install its own administrative facets.

When the node is ready to deactivate a server, it invokes the `shutdown` operation on the server's `Ice::Process` proxy. If the server does not terminate in a timely manner, the node asks the operating system to terminate the process. Each server can be configured with its own deactivation timeout. If no timeout is configured, the node uses the value of the property `IceGrid.Node.WaitTime`, which defaults to `60` seconds.

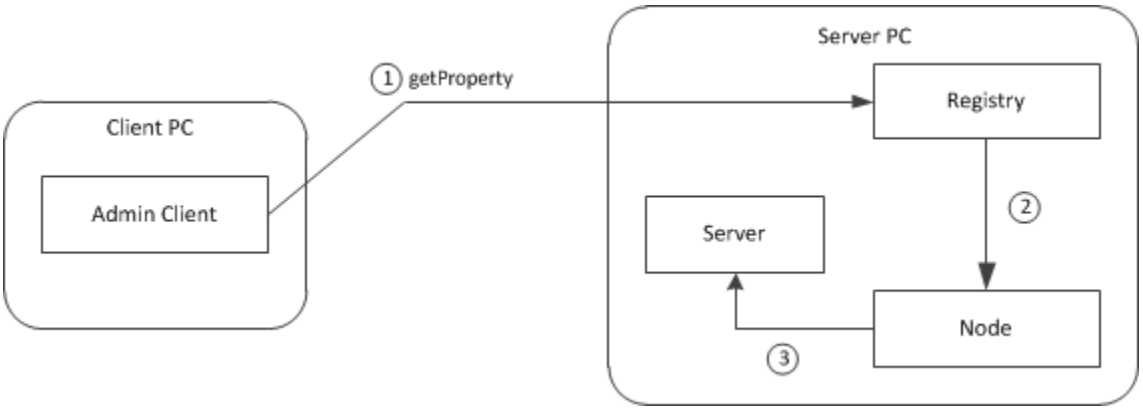
If a server does not register an `Ice::Process` proxy, the IceGrid node cannot request a graceful termination and must resort instead to a more drastic, and potentially harmful, alternative by asking the operating system to terminate the server's process. On Unix, the node sends the `SIGTERM` signal to the process and, if the server does not terminate within the deactivation timeout period, sends the `SIGKILL` signal.

On Windows, the node first sends a `Ctrl+Break` event to the server and, if the server does not stop within the deactivation timeout period, terminates the process immediately.

Servers that disable the `Process` facet can install a signal handler in order to intercept the node's notification about pending deactivation. However, we recommend that servers be allowed to use the `Process` facet when possible.

Routing Administrative Requests

IceGrid defaults to using the local host interface when defining the endpoints of a deployed server's administrative object adapter. This configuration allows local clients such as the IceGrid node to access the server's `admin` object while preventing direct invocations from remote clients. A server's `admin` object may still be accessed remotely, but only by clients that establish an IceGrid administrative session. To facilitate these requests, IceGrid uses an intermediary object that relays requests to the server via its node. For example, the following figure illustrates the path of a `getProperty` invocation:



Routing for administrative requests on a server.

Obtaining a Proxy

During an administrative session, a client has two ways of obtaining the intermediary proxy for a server's `admin` object:

Slice

```
module IceGrid
{
    interface Admin
    {
        idempotent string getServerAdminCategory();
        idempotent Object* getServerAdmin(string id)
            throws ServerNotExistException, NodeUnreachableException,
DeploymentException;
        // ...
    }
}
```

If the client wishes to construct the proxy itself and already knows the server's ID, the client need only modify the proxy of the `IceGrid::Admin` object with a new identity. The identity's category must be the return value of `getServerAdminCategory`, while its name is the ID of the desired server. The example below demonstrates how to create the proxy and access the `Properties` facet of a server:

C++11 C++98

```
shared_ptr<IceGrid::AdminSessionPrx> session = ...;
auto admin = session->getAdmin();
Ice::Identity serverAdminId;
serverAdminId.category = admin->getServerAdminCategory();
serverAdminId.name = "MyServerId";
auto props =
Ice::checkedCast<Ice::PropertiesAdminPrx>(admin->ice_identity(serverAdm
inId), "Properties");
```

```
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::Identity serverAdminId;
serverAdminId.category = admin->getServerAdminCategory();
serverAdminId.name = "MyServerId";
Ice::PropertiesAdminPrx props =
Ice::PropertiesAdminPrx::checkedCast(admin->ice_identity(serverAdminId)
, "Properties");
```

Alternatively, the `getServerAdmin` operation returns a proxy that refers to the `admin` object of the given server. This operation performs additional validation and therefore may raise one of the exceptions shown in its signature above.

A client can also obtain an intermediary proxy to the admin object of an IceGrid node or IceGrid registry replica by calling `getNodeAdmin` or `getRegistryAdmin`:

Slice

```
module IceGrid
{
    interface Admin
    {
        idempotent Object* getNodeAdmin(string name)
            throws NodeNotExistException, NodeUnreachableException;

         idempotent Object* getRegistryAdmin(string name)
            throws RegistryNotExistException;
        // ...
    }
}
```

The name parameter corresponds to the node name or registry replica name. There is no operation comparable to `getServerAdminCategory` for IceGrid nodes and IceGrid registry replicas.

Callbacks without Glacier2

IceGrid also supports the relaying of callback requests from a back-end server to an administrative client over the client's existing connection to the registry, which is especially important for a client using a network port that is forwarded by a firewall or protected by a secure tunnel.

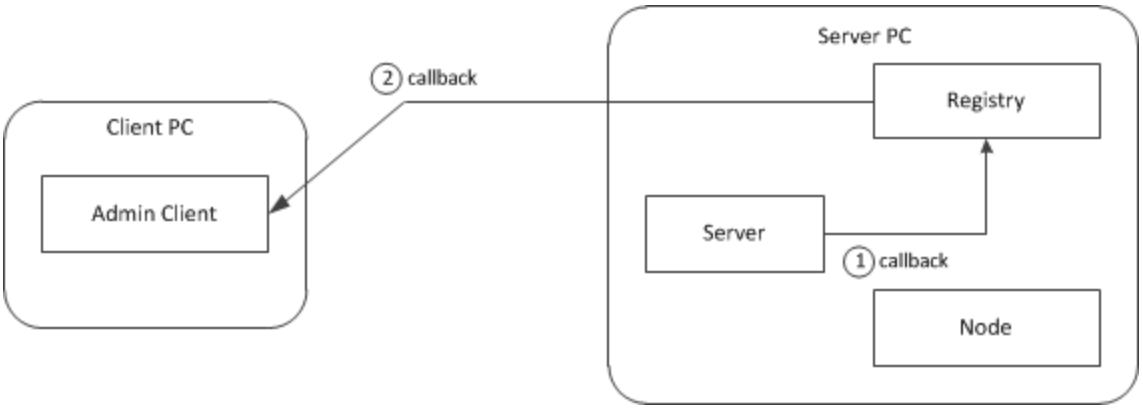
For this mechanism to work properly, a client that established its administrative session directly with IceGrid and not via a Glacier2 router must take additional steps to ensure that the proxies for its callback objects contain the proper identities and endpoints. The `IceGrid::AdminSession` interface provides an operation to help with the client's preparations:

Slice

```
module IceGrid
{
    interface AdminSession ...
    {
        idempotent Object* getAdminCallbackTemplate();
        // ...
    }
}
```

As its name implies, the `getAdminCallbackTemplate` operation returns a *template proxy* that supplies the identity and endpoints a client needs to configure its callback objects. The information contained in the template proxy is valid for the lifetime of the administrative session. This operation returns a null proxy if the client's administrative session was established via a Glacier2 router, in which case the client should use the callback strategy described in the next section instead.

The endpoints contained in the template proxy are those of an object adapter in the IceGrid registry. The client must transfer these endpoints to the proxies for its callback objects so that callback requests from a server are sent first to IceGrid and then relayed over a bidirectional connection to the client, as shown below:



Routing for callback requests from a server.

Here is the complete list of steps:

1. Invoke `getAdminCallbackTemplate` to obtain the template proxy.
2. Extract the category from the template proxy's identity and use it in all callback objects.
3. Extract the endpoints from the template proxy and use them to establish the published endpoints of the callback object adapter.
4. Create the callback object adapter and associate it with the administrative session's connection, thereby establishing a bidirectional connection with IceGrid.
5. Add servants to the callback object adapter.

As an example, let us assume that we have deployed an IceBox server with the server id `icebox1` and our objective is to register a `ServiceObserver` callback that monitors the state of the IceBox services. The first step is to obtain a proxy for the administrative facet named `IceBox.ServiceManager`:

C++11 C++98

```
shared_ptr<IceGrid::AdminSessionPrx> session = ...;
auto admin = session->getAdmin();
auto obj = admin->getServerAdmin("icebox1");
auto svcmgr = Ice::checkedCast<IceBox::ServiceManagerPrx>(obj, "IceBox.
ServiceManager");
```

```
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::ObjectPrx obj = admin->getServerAdmin("icebox1");
IceBox::ServiceManagerPrx svcmgr = IceBox::ServiceManagerPrx::checkedCa
st(obj, "IceBox.ServiceManager");
```

Next, we retrieve the template proxy and compose the published endpoints for our callback object adapter:

C++11 C++98

```
auto tmpl = admin->getAdminCallbackTemplate();
auto endpts = tmpl->ice_getEndpoints();
string publishedEndpoints;
for(auto p = endpts.begin(); p != endpts.end(); ++p)
{
    if(p == endpts.begin())
    {
        publishedEndpoints = (*p)->toString();
    }
    else
    {
        publishedEndpoints += ":" + (*p)->toString();
    }
}
communicator->getProperties()->setProperty("CallbackAdapter.PublishedEn
dpoints", publishedEndpoints);
```

```
Ice::ObjectPrx tmpl = admin->getAdminCallbackTemplate();
Ice::EndpointSeq endpts = tmpl->ice_getEndpoints();
string publishedEndpoints;
for(Ice::EndpointSeq::const_iterator p = endpts.begin();
p != endpts.end(); ++p)
{
    if(p == endpts.begin())
    {
        publishedEndpoints = (*p)->toString();
    }
    else
    {
        publishedEndpoints += ":" + (*p)->toString();
    }
}
communicator->getProperties()->setProperty("CallbackAdapter.PublishedEn
dpoints", publishedEndpoints);
```

The final steps involve creating the callback object adapter, adding a servant, establishing the bidirectional connection and registering our callback with the service manager:

C++11 C++98

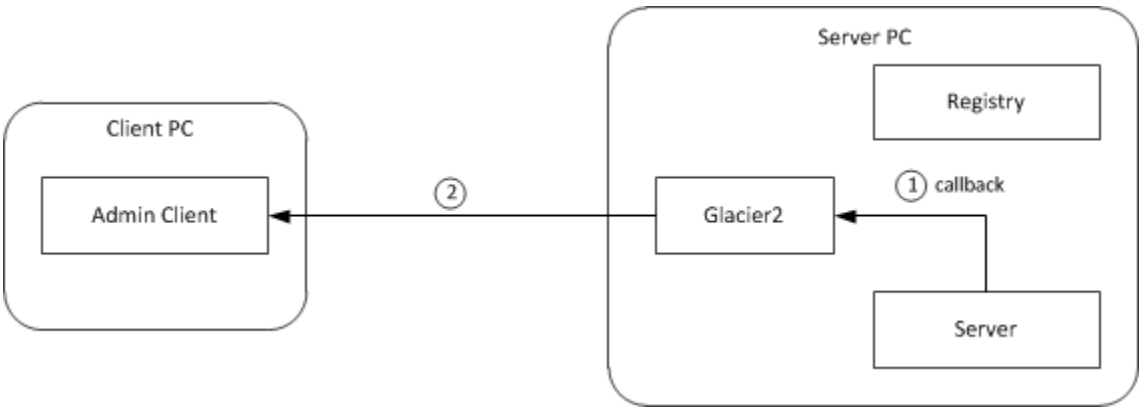
```
auto callbackAdapter =
communicator->createObjectAdapter("CallbackAdapter");
Ice::Identity cbid;
cbid.category = tmpl->ice_getIdentity().category;
cbid.name = "observer";
auto obs = make_shared<ObserverI>();
auto cbobj = callbackAdapter->add(obs, cbid);
auto cb = Ice::uncheckdCast<IceBox::ServiceObserverPrx>(cbobj);
callbackAdapter->activate();
session->ice_getConnection()->setAdapter(callbackAdapter);
svcmgr->addObserver(cb);
```

```
Ice::ObjectAdapterPtr callbackAdapter =
communicator->createObjectAdapter("CallbackAdapter");
Ice::Identity cbid;
cbid.category = tmpl->ice_getIdentity().category;
cbid.name = "observer";
IceBox::ServiceObserverPtr obs = new ObserverI;
Ice::ObjectPrx cbobj = callbackAdapter->add(obs, cbid);
IceBox::ServiceObserverPrx cb =
IceBox::ServiceObserverPrx::uncheckedCast(cbobj);
callbackAdapter->activate();
session->ice_getConnection()->setAdapter(callbackAdapter);
svcmgr->addObserver(cb);
```

At this point the client is ready to receive callbacks from the IceBox server whenever one of its services changes state.

Callbacks with Glacier2

A client that creates an administrative session via a Glacier2 router already has a bidirectional connection over which callbacks from administrative facets are relayed. The flow of requests is shown in the illustration below, which presents a simplified view with the router and IceGrid services all running on the same host.



Routing for callback requests from a server.

To prepare for receiving callbacks, the client must perform the same steps as for any router client:

1. Obtain a proxy for the router.
2. Retrieve the category to be used in callback objects.
3. Create the callback object adapter and associate it with the router, thereby establishing a bidirectional connection.
4. Add servants to the callback object adapter.

Repeating the example from the previous section, we assume that we have deployed an IceBox server with the server ID `icebox1` and our objective is to register a `ServiceObserver` callback that monitors the state of the IceBox services. The first step is to obtain a proxy for the administrative facet named `IceBox.ServiceManager`:

C++11 C++98

```
shared_ptr<IceGrid::AdminSessionPrx> session = ...;
auto admin = session->getAdmin();
auto obj = admin->getServerAdmin("icebox1");
auto svcmgr =
Ice::checkedCast<IceBox::ServiceManagerPrx>(obj, "IceBox.ServiceManager
");
```

```
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::ObjectPrx obj = admin->getServerAdmin("icebox1");
IceBox::ServiceManagerPrx svcmgr =
IceBox::ServiceManagerPrx::checkedCast(obj, "IceBox.ServiceManager");
```

Now we are ready to create the object adapter and register the observer:

C++11 C++98

```
auto router = communicator->getDefaultRouter();
auto callbackAdapter =
communicator->createObjectAdapterWithRouter("CallbackAdapter", router);
Ice::Identity cbid;
cbid.category = router->getCategoryForClient();
cbid.name = "observer";
auto obs = make_shared<ObserverI>();
auto cbobj = callbackAdapter->add(obs, cbid);
auto cb = Ice::uncheckedCast<IceBox::ServiceObserverPrx>(cbobj);
callbackAdapter->activate();
svcmgr->addObserver(cb);
```

```
Ice::RouterPrx router = communicator->getDefaultRouter();
Ice::ObjectAdapterPtr callbackAdapter =
communicator->createObjectAdapterWithRouter("CallbackAdapter", router);
Ice::Identity cbid;
cbid.category = router->getCategoryForClient();
cbid.name = "observer";
IceBox::ServiceObserverPtr obs = new ObserverI;
Ice::ObjectPrx cbobj = callbackAdapter->add(obs, cbid);
IceBox::ServiceObserverPrx cb =
IceBox::ServiceObserverPrx::uncheckedCast(cbobj);
callbackAdapter->activate();
svcmgr->addObserver(cb);
```

At this point the client is ready to receive callbacks from the IceBox server whenever one of its services changes state.

Using the Administrative Facility in IceGrid Utilities

This section discusses the ways in which the IceGrid utilities make use of the administrative facility.

Properties

The command line and graphical utilities allow you to explore the configuration properties of a server or service.

One property in particular, `BuildId`, is given special consideration by the graphical utility. Although it is not used by the Ice run time, the `BuildId` property gives you the ability to describe the build configuration of your application. The property's value is shown by the graphical utility in its own field in the attributes of a server or service, as well as in the list of properties. You can also retrieve the value of this property using the command-line utility with the following statement:

```
> server property MyServerId BuildId
```

Or, for an IceBox service, with this command:

```
> service property MyServerId MyService BuildId
```

The utilities use the `Properties` facet to access these properties, via a proxy obtained as described above.

Administering IceBox Services

IceBox provides an administrative facet that implements the `IceBox::ServiceManager` interface, which supports operations for stopping an active service, and for starting a service that is currently inactive. These operations are available in both the command line and graphical utilities.

IceBox also defines a `ServiceObserver` interface for receiving callbacks when services are stopped or started. The graphical utility implements this interface so that it can present an updated view of the state of an IceBox server. We presented examples that demonstrate how to register an observer with the IceBox administrative facet.

See Also

- Administrative Facility

Securing IceGrid

IceGrid's registry and node services expose multiple network endpoints that a malicious client could use to gain access to IceGrid functionality and interfere with deployed applications. This presents a significant security risk in network environments that are exposed to untrusted clients. For example, a malicious client could connect to a node and use IceGrid's internal interfaces to deploy and run its own server executable.

This page describes the steps you can take to secure your IceGrid application.

On this page:

IceGrid Security Overview

Using a firewall is one way to prevent unauthorized use of IceGrid's facilities. Another solution is to use IceSSL: you can generate SSL certificates for each component and configure them to trust and accept connections only from other authorized components. The remainder of this section discusses the IceSSL solution but also provides useful information for those interested in securing IceGrid with a firewall.

To restrict access using IceSSL, we need to establish trust relationships between IceGrid registry replicas, nodes, and deployed servers. IceSSL allows us to do this using configuration properties. The trust relationships are based on the information contained in SSL certificates.

There are several possible strategies for generating certificates. At a minimum you will need the following:

- one certificate for all of the registries
- one certificate for all of the nodes
- one certificate for all of the servers managed by IceGrid

The certificates that you generate for registries and nodes should be protected with a password to ensure that only privileged users can start these services. However, we do not recommend using a password to protect the certificate for deployed servers because it would need to be specified in clear text in each server's configuration (servers that are activated by IceGrid must not prompt for a password). Furthermore, this password might appear in multiple places, such as an XML descriptor file, the IceGrid registry database, and property files generated by IceGrid nodes. The complexity involved in protecting access to every file that contains a clear text password could be overwhelming. Instead, we recommend that you protect access to the server certificate using file system permissions.

Depending on your organization and the roles of each person that uses IceGrid, you may decide to create additional certificates. For example, you might create a unique certificate for each IceGrid node instance if you deploy nodes on end-user machines and wish to configure the IceGrid registry to authorize connections only from the nodes of trusted users.

You can use the `iceca` script to establish a certificate authority and generate certificates. The sections that follow describe the interactions between the registry, node, and servers, and show how to configure IceSSL to restrict access to trusted peers. For the purposes of this discussion, we assume that the SSL certificates use the common names shown below:

- `IceGrid Registry`
- `IceGrid Node`
- `Server`

The Ice distribution includes a C++ example that demonstrates how to configure a secure IceGrid deployment in the `demo/IceGrid/secure` subdirectory. This example includes a script to generate certificates for a registry, a node, a Glacier2 router, and a server. For more information, see the `README` file provided with the example.

Understanding the Registry Endpoints

The IceGrid registry has three mandatory endpoints representing the client, server, and internal endpoints. The registry also has two optional endpoints (the session manager and administrative session manager endpoints) that are only useful when accessing IceGrid via Glacier2.

Client Endpoint

The registry client endpoint is used by Ice applications that create client sessions in order to use the resource allocation facility. It is also used by administrative clients that create sessions for managing the registry. Finally, the client endpoint is used by Ice applications that use the `IceGrid::Query` interface or resolve indirect proxies via the IceGrid locator.

Two distinct permission verifiers authorize the creation of client sessions and administrative sessions. The remaining functionality available via the client endpoint, such as resolving objects and object adapters using the `IceGrid::Query` interface or the Ice locator mechanism, is accessible to any client that is able to connect to the client endpoint.

It is safe to use an insecure transport for the client endpoint if it is only being used for locator queries. However, you should use a secure transport if you have enabled client and administrative sessions (by configuring the appropriate permission verifiers). Creating a session over an insecure transport poses a security risk because the user name and password are sent in clear text over the network.

If you include secure and insecure transports in the registry's client endpoints, you should ensure that applications that need to authenticate with IceGrid permission verifiers use a secure transport.

It is not necessary to restrict SSL access to the client endpoints (using the property `IceSSL.TrustOnly.Server.IceGrid.Registry.Client`) as long as you use client and administrative permission verifiers for authentication. This property is only useful for restricting access to client and administrative sessions when using null permission verifiers. Note however that if both client and administrative sessions are enabled, you will only be able to restrict access to one set of clients since you cannot distinguish clients that create client sessions from clients that create administrative sessions.

Server Endpoint

Ice servers use the registry's server endpoint to register their object adapter endpoints and send information to administrative clients connected via the registry.

Securing this endpoint with IceSSL is necessary to prevent a malicious program from potentially hijacking a server by registering its endpoints first. The property definition shown below demonstrates how to limit access to this endpoint to trusted Ice servers:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.Server=CN="Server"
```

Internal Endpoint

IceGrid nodes and registry replicas use the internal endpoint to communicate with the registry. For example, nodes connect to the internal endpoint of each active registry, and registry slaves establish a session with their master via this endpoint.

The internal endpoint must be secured with IceSSL to prevent malicious Ice applications from gaining access to sensitive functionality that is intended to be used only by nodes and registry replicas. You can restrict access to this endpoint with the following property:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.Internal=CN="IceGrid
Node";CN="IceGrid Registry"
```

Session Manager Endpoint

The session manager endpoint is used by Glacier2 to create IceGrid client sessions. The functionality exposed by this endpoint is unrestricted so you must either secure it or disable it (this endpoint is disabled by default). The property shown below demonstrates how to configure IceSSL so that only Glacier2 routers are accepted by this endpoint:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.SessionManager=CN="Glacier2
Router Client"
```

In this example, `Glacier2 Router Client` is the common name of the Glacier2 router used by clients to create IceGrid client sessions.

Administrative Session Manager Endpoint

Glacier2 routers use the registry's administrative session manager endpoint to create IceGrid administrative sessions. The functionality exposed by this endpoint is unrestricted, so you must either secure it or disable it (this endpoint is disabled by default). The property shown below demonstrates how to configure IceSSL so that only Glacier2 routers are accepted by this endpoint:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.AdminSessionManager=CN="Glacie
r2 Router Admin"
```

In this example, `Glacier2 Router Admin` is the common name of the Glacier2 router used by clients to create IceGrid administrative sessions. Note that if you use a single Glacier2 router instance for both client and administrative sessions, you will need to use the same common name to restrict access to both session manager endpoints:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.SessionManager=CN="Glacier2
Router Client"
IceSSL.TrustOnly.Server.IceGrid.Registry.AdminSessionManager=CN="Glacie
r2 Router Client"
```

IceLocatorDiscovery Endpoint

The registry (including all replicas) listens by default for UDP multicast requests from IceLocatorDiscovery clients. Each client request includes a proxy to which the registry sends its response; by default, the client's "response callback" proxy uses a UDP unicast endpoint but it can be configured to use a different transport. The registry invokes on the client's response callback proxy and provides a proxy of its own containing the registry's client endpoint.

Ice does not support a secure multicast transport therefore these discovery requests cannot be encrypted or restricted via trust relationships. You can prevent registries from listening for discovery requests by setting `IceGrid.Registry.Discovery.Enabled`:

```
IceGrid.Registry.Discovery.Enabled=0
```

Outgoing Connections

The registry establishes outgoing connections to other registries and nodes. You should configure the `IceSSL.TrustOnly.Client` property to restrict connections to these trusted peers:

```
IceSSL.TrustOnly.Client=CN="IceGrid Registry";CN="IceGrid Node"
```

The registry can also connect to Glacier2 routers and permission verifier objects. To allow connections to these services, you must include in this property the common names of Glacier2 routers that create client or administrative sessions, as well as the common names of servers that host the permission verifier objects.

Understanding the Node Endpoints

An IceGrid node has only one endpoint, which is used for internal communications with the registry. As a result, it should be configured to accept connections only from IceGrid registries:

```
IceSSL.TrustOnly.Server=CN="IceGrid Registry"
```

A node also establishes outgoing connections to the registry's internal endpoint, as well as the `Ice.Admin` endpoint of deployed servers. You should configure the `IceSSL.TrustOnly.Client` property as shown below to verify the identity of these peers:

```
IceSSL.TrustOnly.Client=CN="Server";CN="IceGrid Registry"
```

Understanding the Administrative Endpoints with IceGrid

By default, IceGrid sets the endpoints of a deployed server's `Ice.Admin` adapter to `tcp -h 127.0.0.1`. This setting is already quite secure because it only accepts connections from processes running on the same host. However, since you already need to configure IceSSL so that a server can authenticate with the IceGrid registry (servers connect to the registry to register their endpoints), you might as well use a secure endpoint for the `Ice.Admin` adapter and configure it to accept connections only from IceGrid nodes:

```
IceSSL.TrustOnly.Server.Ice.Admin=CN="IceGrid Node"
```

This is only necessary if the `Ice.Admin` endpoint is enabled (which it is by default).

You can also set the `IceSSL.TrustOnly.Client` property so that the server is only permitted to connect to the IceGrid registry:

```
IceSSL.TrustOnly.Client=CN="IceGrid Registry"
```

If your server invokes on other servers, you will need to modify this setting to allow secure connections to them.

See Also

- IceSSL
- Configuring IceSSL
- Setting up a Certificate Authority
- Glacier2 Integration with IceGrid
- Resource Allocation using IceGrid Sessions
- IceGrid Administrative Sessions

- Well-Known Objects
- IceGrid and the Administrative Facility
- Registry Replication
- IceSSL.*

icegridadmin Command Line Tool

The `icegridadmin` utility is a command-line tool for administering an IceGrid domain. Deploying an application with this utility requires an XML file that defines the descriptors.

On this page:

Usage

The IceGrid administration tool supports the following command-line options:

```
Usage: icegridadmin [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-e COMMANDS          Execute COMMANDS.
-d, --debug          Print debug messages.
-s, --server         Start icegridadmin as a server (to parse XML
files).
-i, --instanceName   Connect to the registry with the given instance
name.
-H, --host           Connect to the registry at the given host.
-P, --port           Connect to the registry running on the given port.
-u, --username       Login with the given username.
-p, --password       Login with the given password.
-S, --ssl            Authenticate through SSL.
-r, --replica NAME   Connect to the replica NAME.
```

The `-e` option causes the tool to execute the given commands and then exit without entering an interactive mode. The `-s` option starts `icegridadmin` in a server mode that supports the `IceGrid::FileParser` interface; a proxy for the object is printed to standard output. If neither `-e` nor `-s` is specified, the tool enters an interactive mode in which you issue commands at a prompt.

To communicate with the IceGrid registry, `icegridadmin` establishes an administrative session. The tool uses SSL authentication if you specify the `-S` option or define its equivalent property `IceGridAdmin.AuthenticateUsingSSL`. Otherwise, `icegridadmin` uses password authentication and prompts you for the username and password if you do not specify them via command-line options or properties. If you want `icegridadmin` to establish its session using a Glacier2 router, define `Ice.Default.Router` appropriately.

If specified, the following command-line options override their property equivalents:

Option	Property
`-i, --instanceName`	`IceGridAdmin.InstanceName`
`-H, --host`	`IceGridAdmin.Host`

`-P, --port`	`IceGridAdmin.Port`
`-u, --username`	`IceGridAdmin.Username`
`-p, --password`	`IceGridAdmin.Password`
`-S, --ssl`	`IceGridAdmin.AuthenticateUsingSSL`
`-r, --replica`	`IceGridAdmin.Replica`

`icegridadmin` determines its target registry as follows:

1. Connect via a Glacier2 router if `Ice.Default.Router` is defined.
2. Otherwise, if `Ice.Default.Locator` is defined, connect to the specified registry.
3. Otherwise, if a host is defined via the `-H` or `--host` options or the equivalent property, connect to the registry at the specified host. If no port is defined via the `-P` or `--port` options or the equivalent property, `icegridadmin` uses the standard IceGrid TCP (4061) or SSL (4062) port.
4. Otherwise, `icegridadmin` attempts to locate a registry by issuing a UDP multicast discovery request. (`icegridadmin` does not use the IceGridDiscovery plug-in.) If the tool discovers more than one registry, it presents a list and asks you to select one.

> See IceGridAdmin.* for more information on the configuration properties supported by `icegridadmin`.

Once it has connected to the registry and successfully established a session, `icegridadmin` displays its command prompt. The `help` command displays the following usage information:

- `help`
  Print this message.
- `exit`, `quit`
  Exit this program.
- `CATEGORY help`
  Print the help section of the given `CATEGORY`
- `COMMAND help`
  Print the help of the given `COMMAND`.

The tool's commands are organized by category. The supported command categories are shown below:

- `application`
- `node`
- `registry`
- `server`
- `service`
- `adapter`
- `object`
- `server template`
- `service template`

You can obtain more information about each category using the `help` command:

```
>>> application help
```

Application Commands

- `application add [-n | --no-patch] DESC [TARGET ... ] [NAME=VALUE ... ]`
  Add applications described in the XML descriptor file `DESC`. If specified the optional targets are deployed. Variables are defined using the `NAME=VALUE` syntax. The application is automatically patched unless the `-n` or `--no-patch` option is used to disable it.
- `application remove NAME`
  Remove the application named `NAME`.

- `application describe NAME`
  Describe the application named `NAME`.
- `application diff [-s | --servers] DESC [TARGET ...] [NAME=VALUE ...]`
  Print the differences between the application in the XML descriptor file `DESC` and the current deployment. If `-s` or `--servers` is specified, print the the list of servers affected by the differences. Variables are defined using the `NAME=VALUE` syntax.
- `application update [-n | --no-restart] DESC [TARGET ...] [NAME=VALUE ...]`
  Update the application in the XML descriptor file `DESC`. If `-n` or `--no-restart` is specified, the update will fail if it would require restarting one or more servers. Variables are defined using the `NAME=VALUE` syntax. Use the `diff --servers` command to discover which servers would be affected by an update, including those that would require a restart.
- `application patch [-f | --force] NAME`
  Patch the application named `NAME`. If `-f` or `--force` is specified, IceGrid will first shut down any servers that depend on the data to be patched.
- `application list`
  List all deployed applications.

Node Commands

- `node list`
  List all registered nodes.
- `node describe NAME`
  Show information about node `NAME`.
- `node ping NAME`
  Ping node `NAME`.
- `node load NAME`
  Print the load of the node `NAME`.
- `node sockets [NAME]`
  Print the number of processor sockets for node `NAME`. If `NAME` is omitted, print the number of processor sockets for each node. (The `IceGrid.Node.ProcessorSocketCount` property allows you to explicitly set this value for systems where the number of sockets cannot be obtained programmatically.)
- `node show [OPTIONS] NAME [log | stderr | stdout]`
  Print the Ice log messages of the node (with `log`), or print the text from the node's standard error or standard output (with `stderr` or `stdout`). The supported options are shown below:
  - `-f`, `--follow`
    With `log`, create a remote logger that prints each new log message.
    With `stderr` or `stdout`, wait for new text to be available in the file where `stderr` or `stdout` is redirected.
  - `-t`, `--tail N`
    Print the last `N` log messages (for `log`) or N lines of text (for `stderr` or `stdout`)
  - `-h`, `--head N`
    Print the first `N` lines of text (invalid option with `log`).
- `node shutdown NAME`
  Shutdown node `NAME`.

Registry Commands

- `registry list`
  List all registered registries.
- `registry describe NAME`
  Show information about registry `NAME`.
- `registry ping NAME`
  Ping registry `NAME`.
- `registry show [OPTIONS] NAME [log | stderr | stdout]`
  Print the Ice log messages of the registry (with `log`), or print the text from the registry's standard error or standard output (with `stde`

rr or stdout). The supported options are shown below:

- `-f`, `--follow`
  With `log`, create a remote logger that prints each new log message.
  With `stderr` or `stdout`, wait for new text to be available in the file where `stderr` or `stdout` is redirected.
- `-t`, `--tail N`
  Print the last `N` log messages (for `log`) or N lines of text (for `stderr` or `stdout`)
- `-h`, `--head N`
  Print the first `N` lines of text (invalid option with `log`).

- `registry shutdown NAME`
  Shutdown registry `NAME`.

Server Commands

- `server list`
  List all registered servers.
- `server remove ID`
  Remove server `ID`.
- `server describe ID`
  Describe server `ID`.
- `server properties ID`
  Get the run-time properties of server `ID`.
- `server property ID NAME`
  Get the run-time property `NAME` of server `ID`.
- `server state ID`
  Get the state of server `ID`.
- `server pid ID`
  Get the process ID of server `ID`.
- `server start ID`
  Start server `ID`.
- `server stop ID`
  Stop server `ID`.
- `server patch ID`
  Patch server `ID`.
- `server signal ID SIGNAL`
  Send `SIGNAL` (such as `SIGTERM` or `15`) to server `ID`.
- `server stdout ID MESSAGE`
  Write `MESSAGE` on server `ID`'s standard output.
- `server stderr ID MESSAGE`
  Write `MESSAGE` on server `ID`'s standard error.
- `server show [OPTIONS] ID [log | stderr | stdout | LOGFILE]`
  Print the Ice log messages of the server (with `log`), or print the text from the server's standard error, standard output or the log file LOGFILE (with `stderr`, `stdout` or `LOGFILE`). The supported options are shown below:
  - `-f`, `--follow`
    With `log`, create a remote logger that prints each new log message.
    With `stderr`, `stdout` and `LOGFILE`, wait for new text to be available in the file.
  - `-t`, `--tail N`
    Print the last `N` log messages (for `log`) or N lines of text (for `stderr`, `stdout` or `LOGFILE`)
  - `-h`, `--head N`
    Print the first `N` lines of text (invalid option with `log`).
- `server enable ID`
  Enable server `ID`.

- `server disable ID`
  Disable server `ID`. A disabled server cannot be started on demand.

Service Commands

- `service start ID NAME`
  Starts service `NAME` in IceBox server `ID`.
- `service stop ID NAME`
  Stops service `NAME` in IceBox server `ID`.
- `service describe ID NAME`
  Describes service `NAME` in IceBox server `ID`.
- `service properties ID NAME`
  Get the run-time properties of service `NAME` from IceBox server `ID`.
- `service property ID NAME PROPERTY`
  Get the run-time property `PROPERTY` of service `NAME` from IceBox server `ID`.
- `service list ID`
  List the services in IceBox server `ID`.

Adapter Commands

- `adapter list`
  List all registered adapters.
- `adapter endpoints ID`
  Show the endpoints of adapter or replica group `ID`.
- `adapter remove ID`
  Remove adapter or replica group `ID`.

Object Commands

The `object` command operates on well-known objects.

- `object add PROXY [TYPE]`
  Add a well-known object to the registry, optionally specifying its type.
- `object remove IDENTITY`
  Remove a well-known object from the registry.
- `object find TYPE`
  Find all well-known objects with the type `TYPE`.
- `object describe EXPR`
  Describe all well-known objects whose stringified identities match the expression `EXPR`. A trailing wildcard is supported in `EXPR`, for example "`object describe Ice*`".
- `object list EXPR`
  List all well-known objects whose stringified identities match the expression `EXPR`. A trailing wildcard is supported in `EXPR`, for example "`object list Ice*`".

Server Template

- `server template instantiate APPLICATION NODE TEMPLATE  [NAME=VALUE ...]`
  Instantiate the requested server template defined in the given application on a node. Variables are defined using the `NAME=VALUE` s yntax.
- `server template describe APPLICATION TEMPLATE`
  Describe a server template`TEMPLATE` from the given application.

Service Template

- `service template describe APPLICATION TEMPLATE`
  Describe a service template`TEMPLATE` from the given application.

See Also

- IceGrid Administrative Sessions
- Glacier2 Integration with IceGrid
- IceGrid XML Features
- Using Descriptor Variables and Parameters
- Application Distribution
- Getting Started with IceGrid
- IceGridAdmin.*
- IceLocatorDiscovery

IceGrid GUI Tool

IceGrid GUI is a graphical client for IceGrid. IceGrid GUI allows you to monitor servers deployed by IceGrid, create new IceGrid applications, deploy these applications, and more.

Topics

Getting Started with IceGrid GUI

This page describes how to launch the IceGrid GUI tool.

On this page:

System Requirements

IceGrid GUI is a Java application supported on a wide range of platforms, including Windows, Linux and macOS.

The minimum requirements for running IceGrid GUI are listed below:

- `icegridgui.jar`, usually installed in the `bin` or `lib` directory of your Ice installation
- Java SE Runtime Environment 8 or later

In order to use IceGrid GUI's metrics graphs feature, you will need the JavaFX Runtime Environment, bundled with recent updates of the Oracle Java SE Runtime Environment on Windows, Linux and macOS.

You can download Oracle Java SE for most platforms from Oracle.

If you want to read IceGrid XML files from IceGrid GUI, you also need to have the `icegridadmin` command-line utility (version 3.7) in your PATH.

Starting IceGrid GUI

On Windows IceGrid GUI can be started by clicking the IceGrid GUI icon in the Start menu. On macOS IceGrid GUI can be started by clicking the IceGrid GUI icon in Finder Applications folder:

On Linux, you can use the `icegridgui` script installed in `/usr/bin`:

```
icegridgui
```

On all platforms, you can also start IceGrid GUI from a terminal by typing:

```
java -jar path-to-icegridgui.jar
```

Command Line Arguments

IceGrid GUI can be configured using Ice properties, and like with most Ice applications, these properties can be set using command-line arguments or a configuration file (or both).

Since IceGrid GUI is an Ice application (a client to the IceGrid registry), setting regular Ice properties can be useful as well. For example, you can set the `Ice.Trace.Network` property to get detailed information about network communications sent to `stderr`.

```
$ icegridgui --Ice.Trace.Network=2
```

There are also a number of properties specific to IceGrid GUI itself, described in IceGridAdmin.*.

If you need to set many properties, it is a good idea to write a configuration file and use the `--Ice.Config` command-line argument to specify the location of this file. For example:

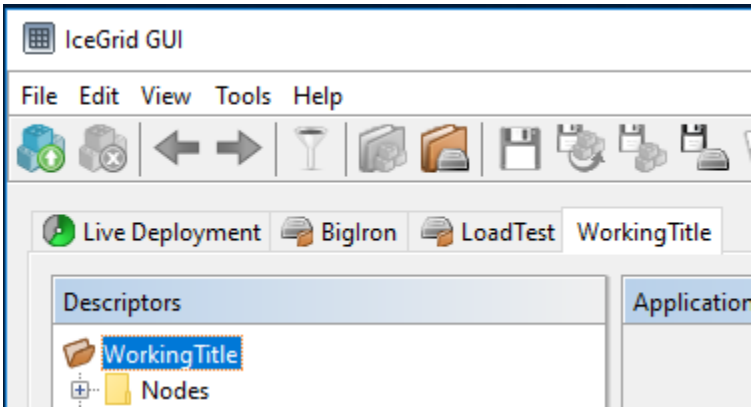
```
> java -jar "C:\Program Files\ZeroC\Ice-3.7.1\bin\icegridgui.jar"
--Ice.Config=icegridgui.cfg
```

Main IceGrid GUI Window

The main IceGrid GUI window allows to navigate between your live deployment and the definitions of several applications.

Tabs

The main IceGrid GUI window shows one or more tabs:

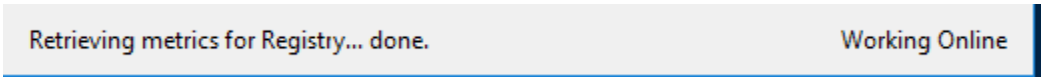


- The Live Deployment tab () displays information about an IceGrid deployment you have logged into. There is always one and only one Live Deployment tab. When you are not connected to an IceGrid deployment, the corresponding pane is empty.
- A Live Application tab () displays application definitions retrieved from the IceGrid registry you are connected to. As long as you do not change anything in the associated pane, IceGrid GUI will keep the information up-to-date. For example if another administrator adds a new server definition in this application definition, it will appear automatically and immediately in this pane.
- A File-Based Application tab () displays application definitions retrieved from an IceGrid XML file.
- An icon-less tab displays the definitions of an application that is not bound to an IceGrid registry or to a file, such as a brand new application. A live application with unsaved modifications also becomes icon-less if the connection to its IceGrid registry is lost.

IceGrid GUI may show any number of application tabs, including none at all.

Status Bar

The status bar at the bottom of the main window shows information about operations performed by IceGrid GUI, or messages received from the IceGrid registry.



See Also

- IceGridAdmin.*
- icegridadmin Command Line Tool

Live Deployment Tab

The Live Deployment tab shows the run-time status and configuration of an existing IceGrid deployment, and allows you to perform various administrative tasks on this deployment, such as:

- start or stop a server
- retrieve metrics associated with a server
- display metrics from one or more servers on a graph
- enable or disable a server
- send signal to a server
- retrieve the log files of a server
- patch an entire application
- register a new "well-known" object with the IceGrid registry
- undeploy a deployed application

Topics

Connection to an IceGrid Registry

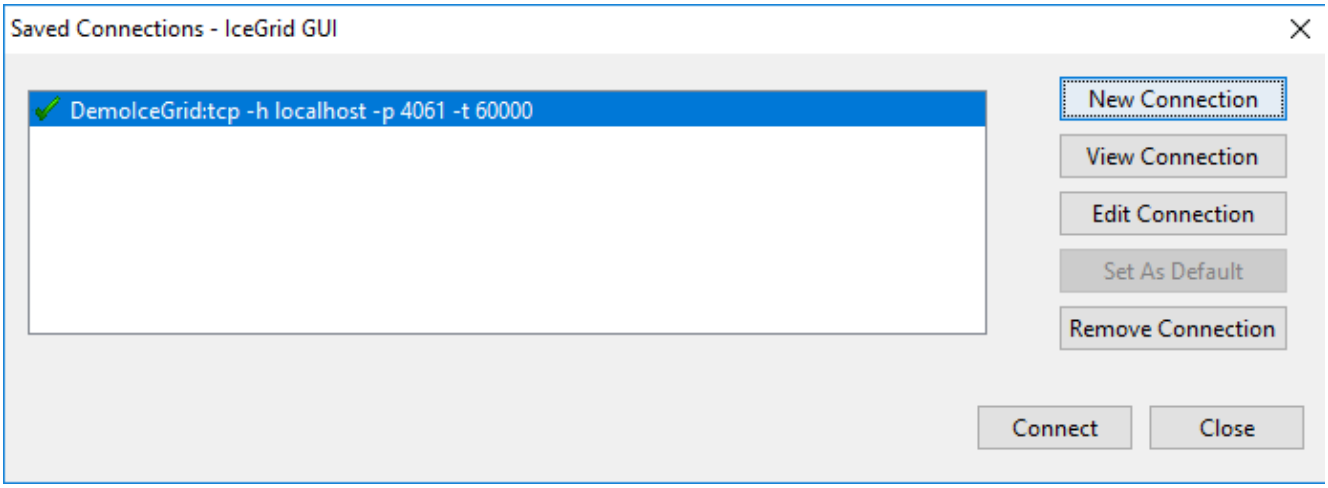
In order to administer or monitor an IceGrid deployment, you need to connect to the IceGrid registry of this deployment. If your IceGrid registry is replicated, you can connect to any replica for monitoring purposes; if you intend to change definitions, for example to describe a new server, you need to connect to the master IceGrid registry. This page describes how to connect to an IceGrid registry.

On this page:

Connecting using a Saved Connection

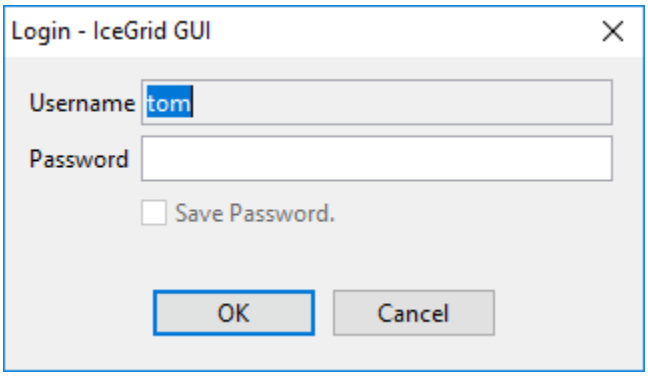
Use `File > Login...` or press the

button to open the `Saved Connections` dialog:



Double-click on the IceGrid registry you want to connect to, or select the IceGrid registry and click on the `Connect` button.

If the connection contains saved credentials, IceGrid GUI will immediately attempt to connect to the selected IceGrid registry with these credentials. Otherwise, it will open a new dialog to request the missing password (two passwords in some cases), such as:



Creating a new Connection

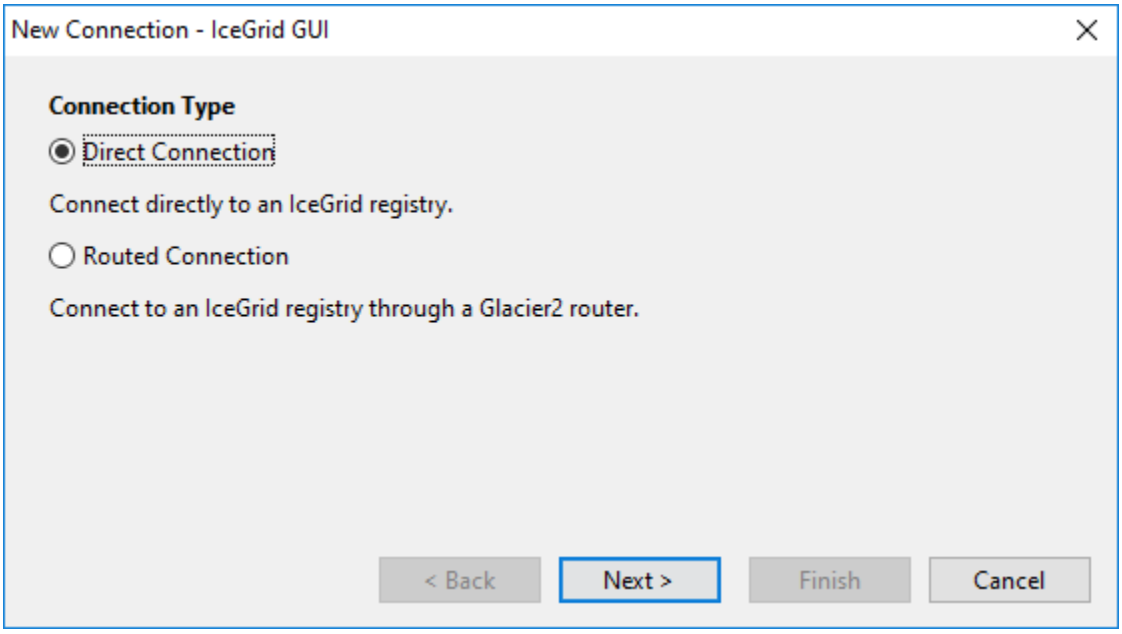
In the `Saved Connections` dialog, click on `New Connection` to open the New Connection wizard.

We describe three typical connection scenarios in the sections below.

TCP Connection to a local IceGrid registry

Let's create a TCP connection to the IceGrid registry running on localhost.

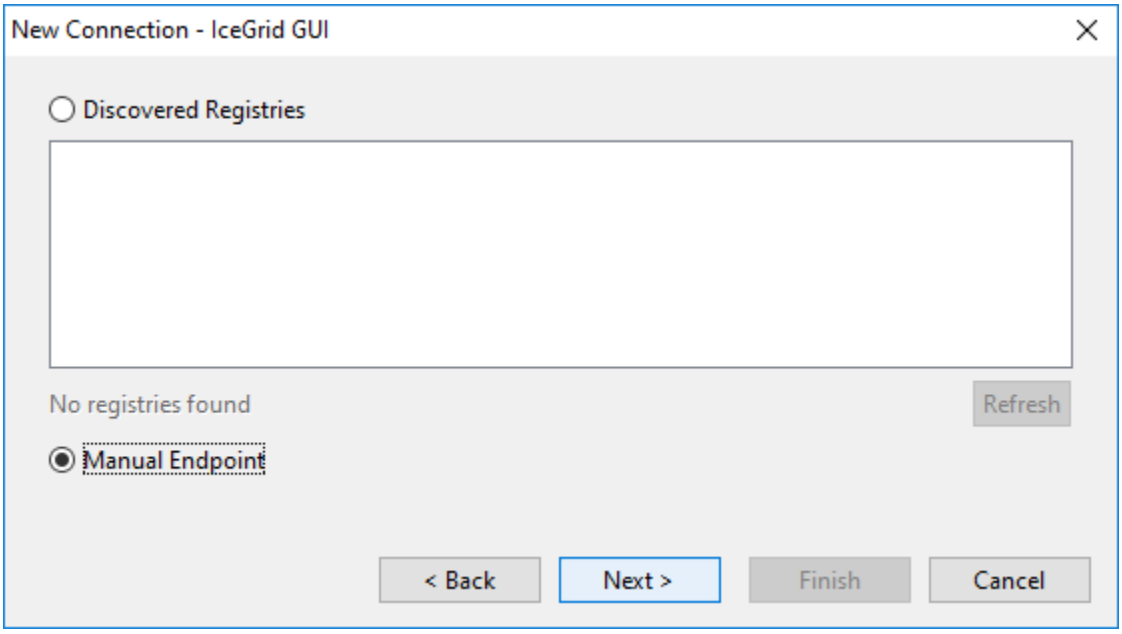
Step 1: Select Direct Connection:



Step 2: Check the box to connect to the master registry:

If you have several replicas this ensures you will always connect to the master registry.

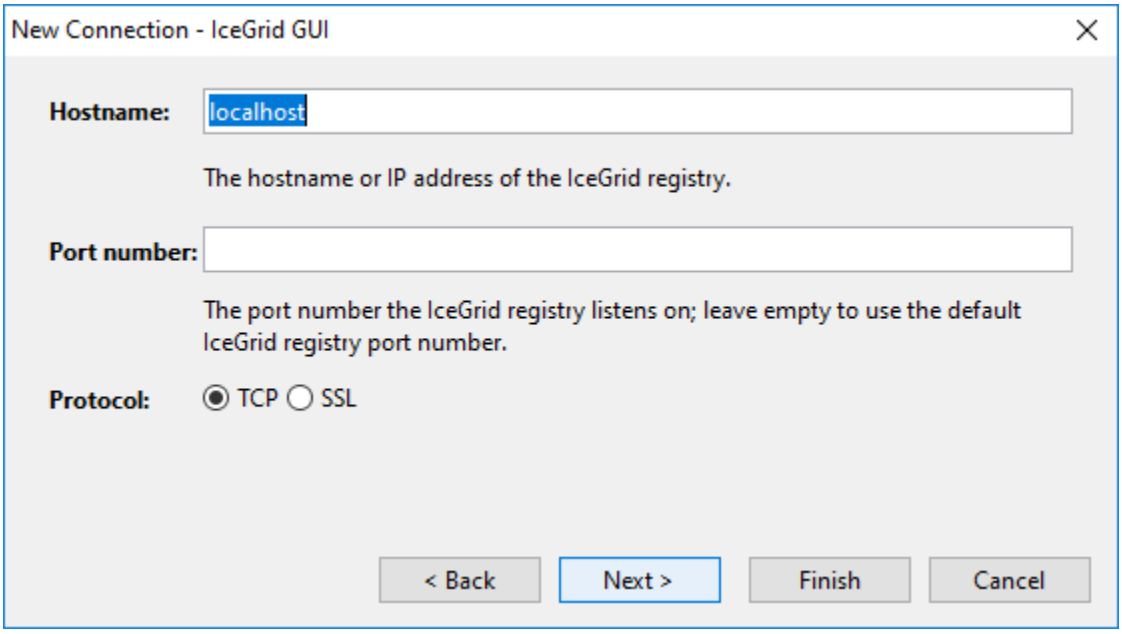
Step 3: Select Manual Endpoint to manually enter address information:



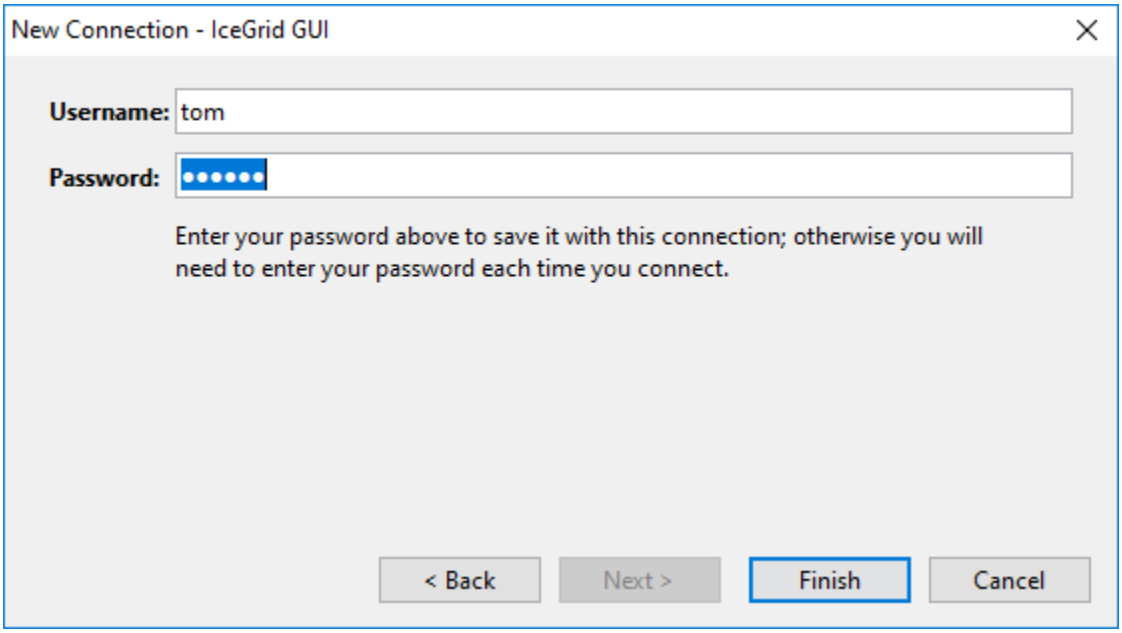
As of Ice 3.6, IceGrid GUI can also discover registries using UDP multicast. Any registries it finds will be shown in the list, or click the Refresh button to search again.

Step 4: Select the first option to enter the addressing information as a hostname and port number:

Step 5: Enter `localhost` for the hostname, leave the port number blank and keep TCP as the protocol:



Step 6: Enter a username and password for this connection:



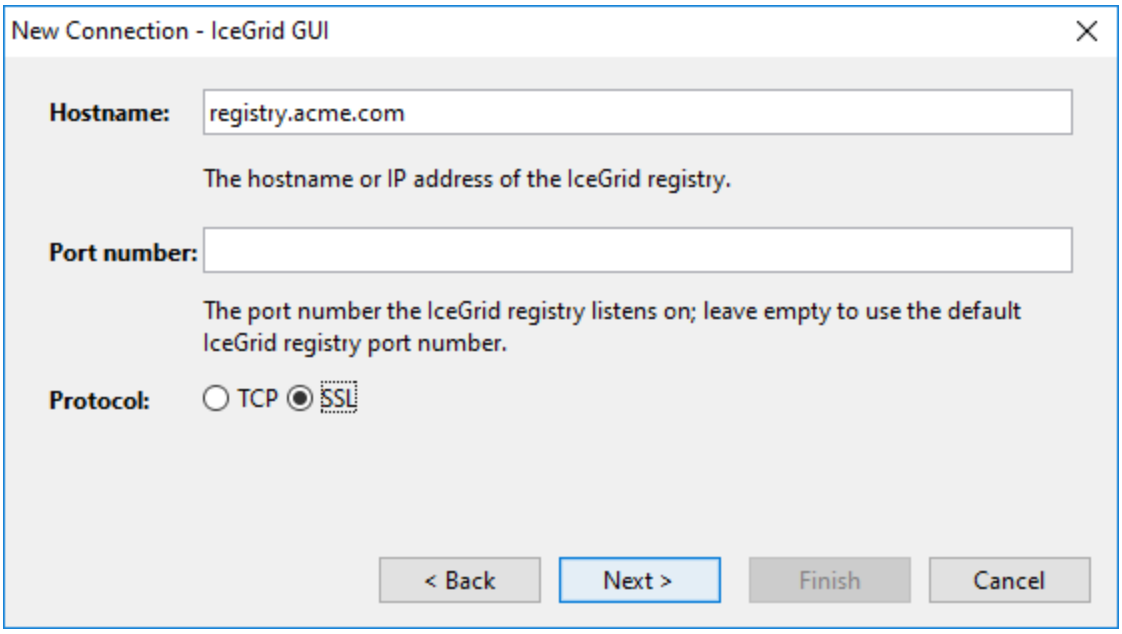
Step 7: Click `Finish` to save the connection; IceGrid GUI then attempts to connect to the IceGrid registry.

SSL Connection to local IceGrid using X.509 Credentials

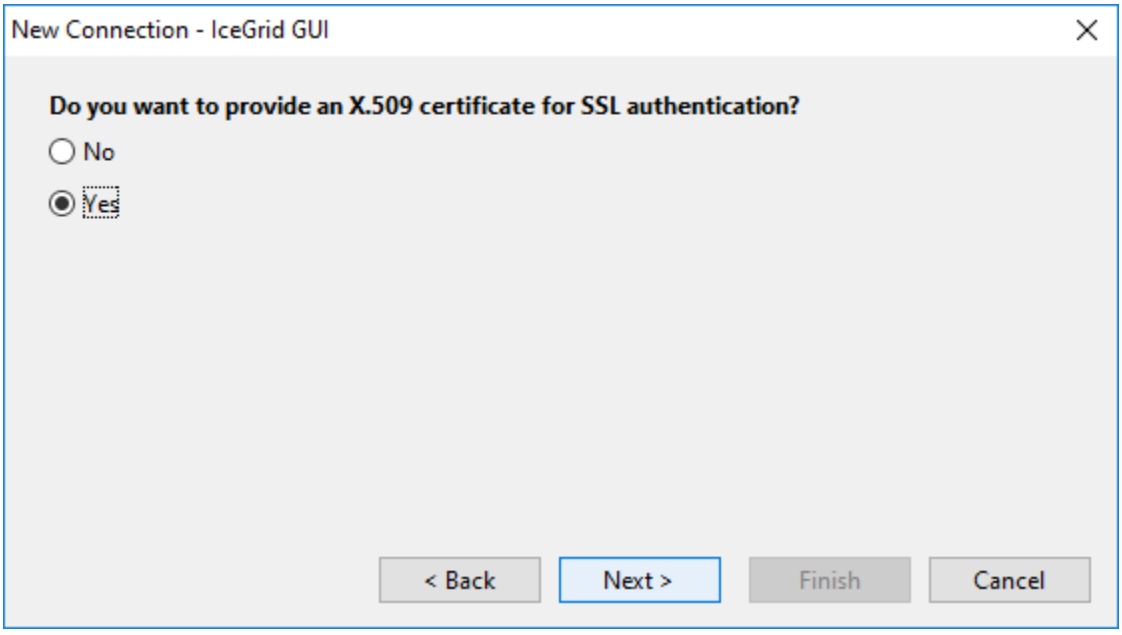
Now let's create a direct SSL connection to an IceGrid registry and authenticate using our X.509 key (also used for SSL authentication). The target IceGrid registry must be configured to accept SSL connections and authentication using SSL credentials; see `IceGrid.Registry.AdminSSLPermissionsVerifier`.

Steps 1 to 4 are identical to the simple TCP connection described above.

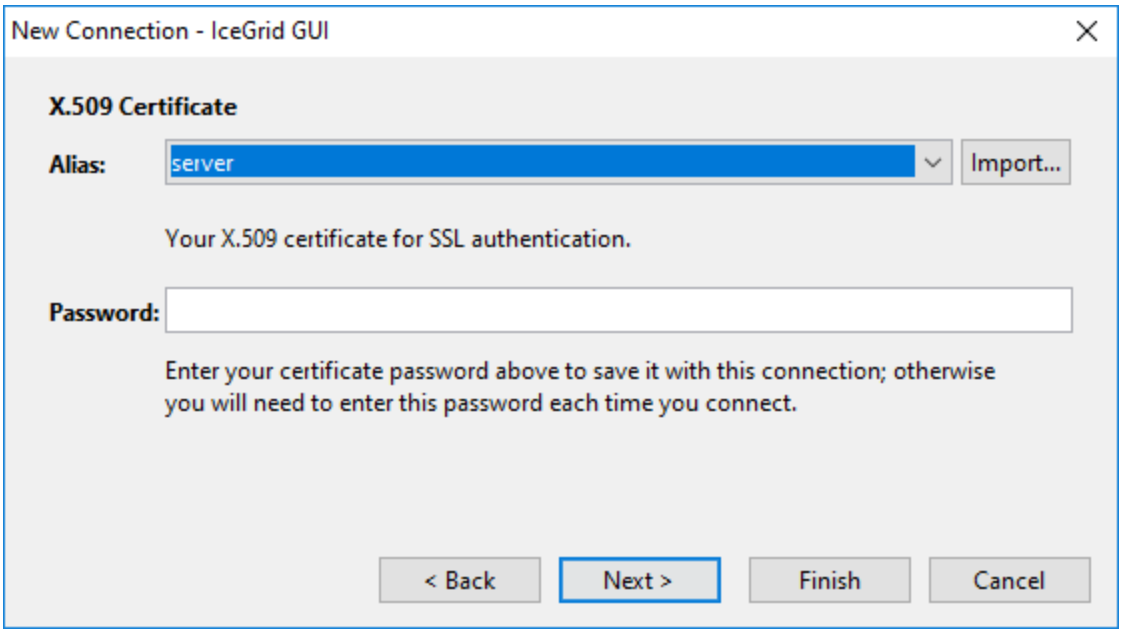
Step 5: Enter the hostname of the IceGrid registry, leave the port number empty to use the default IceGrid port number (4061 for TCP and 4062 for SSL), and select SSL for the protocol:



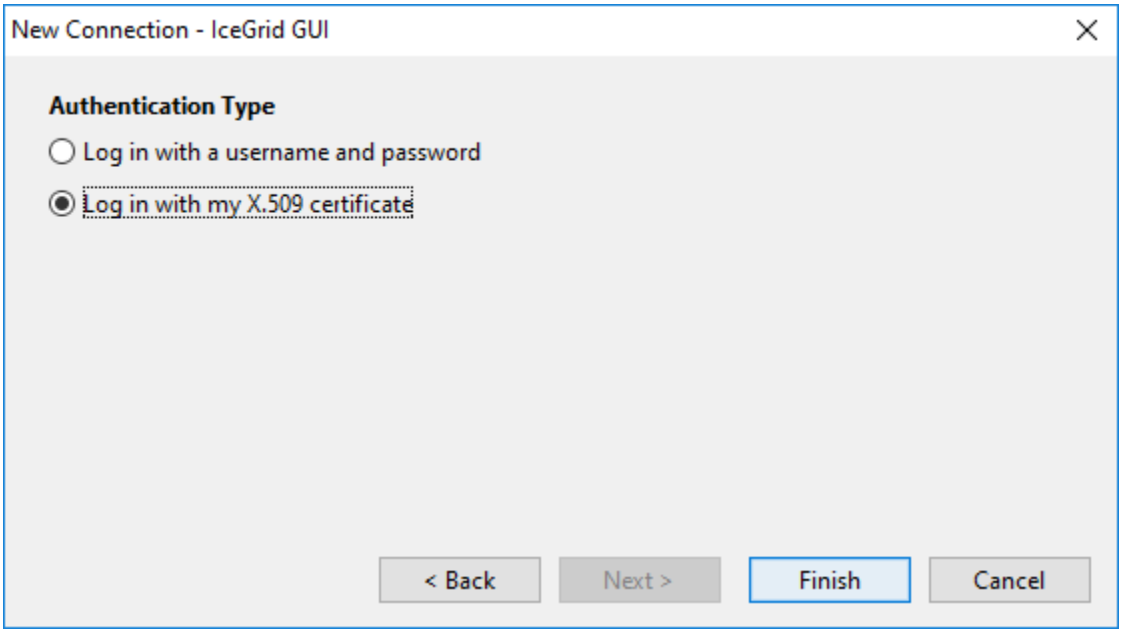
Step 6: Select Yes to provide an X.509 certificate for SSL authentication:



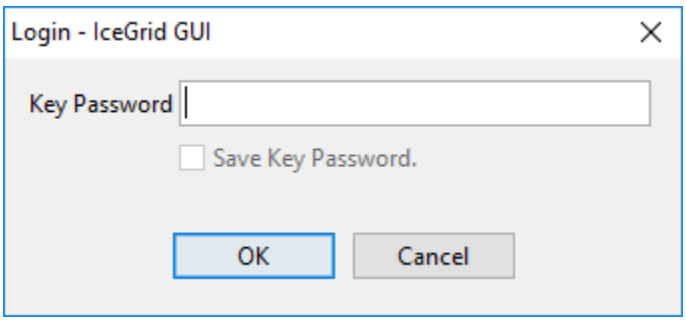
Step 7: Select the X.509 key used for SSL authentication from the Alias drop-down list; this list corresponds to the My Certificates set in the Certificate Manager described in the next section. Click on the `Import...` button to open the Certificate Manager dialog.



Step 8: Choose to use this X.509 certificate (carried through the SSL connection) to authenticate ourselves with the IceGrid registry:



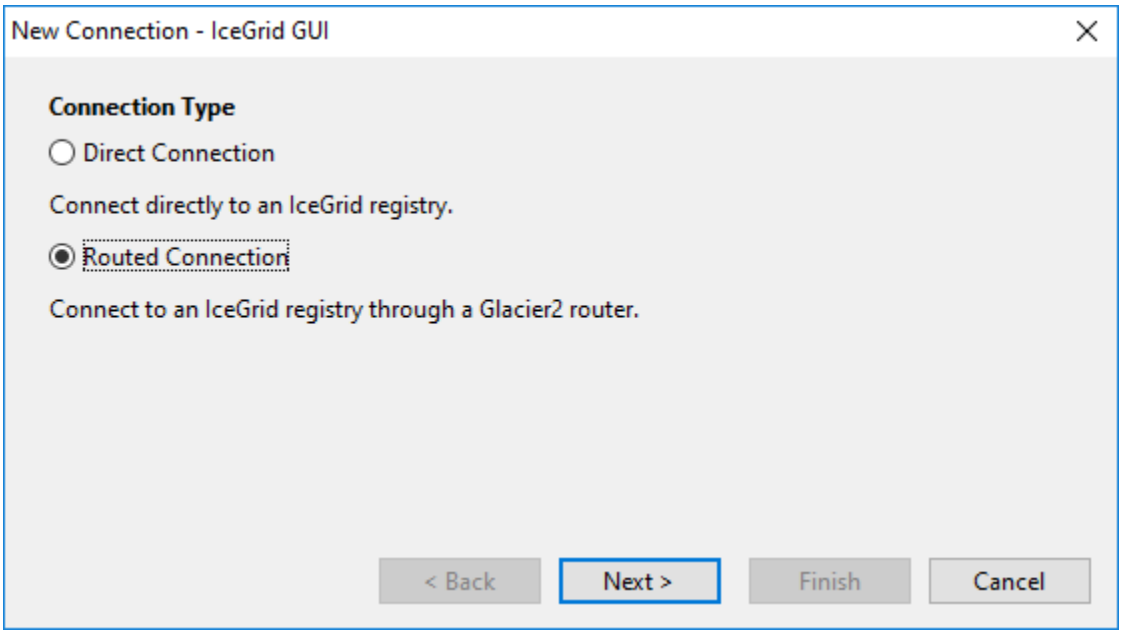
Step 9: Click `Finish` to save the connection and connect to the IceGrid registry. Unless we entered (and therefore saved) the X.509 key password with the connection, we are prompted for this password:



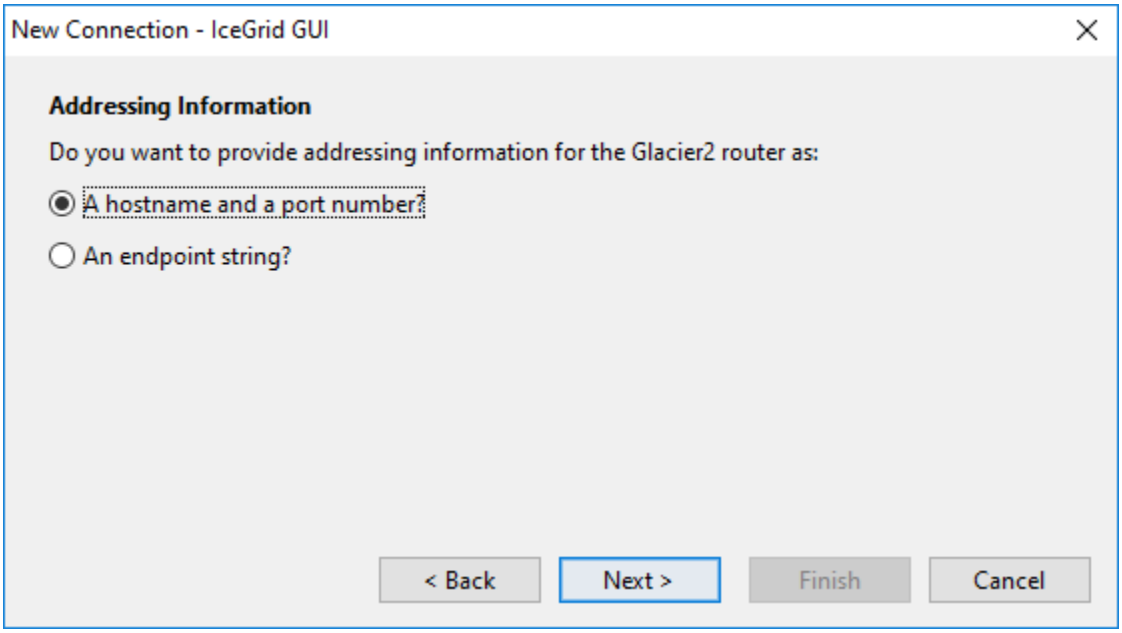
SSL Connection through Glacier2 router

Here we'll connect to an IceGrid registry "behind" a Glacier2 router. In this case, we need to connect to the Glacier2 router and authenticate ourselves with the router. We do not provide any information about the IceGrid registry itself: we will connect to the IceGrid registry identified by the target Glacier2 router configuration. See Glacier2 Integration with IceGrid.

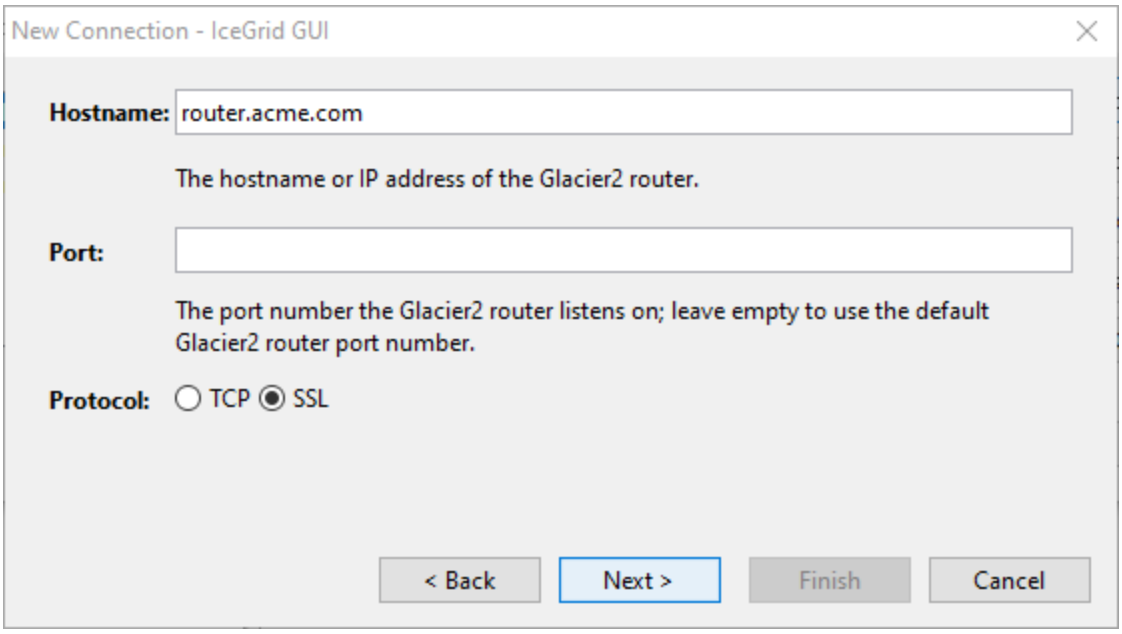
Step 1: Select Routed Connection:



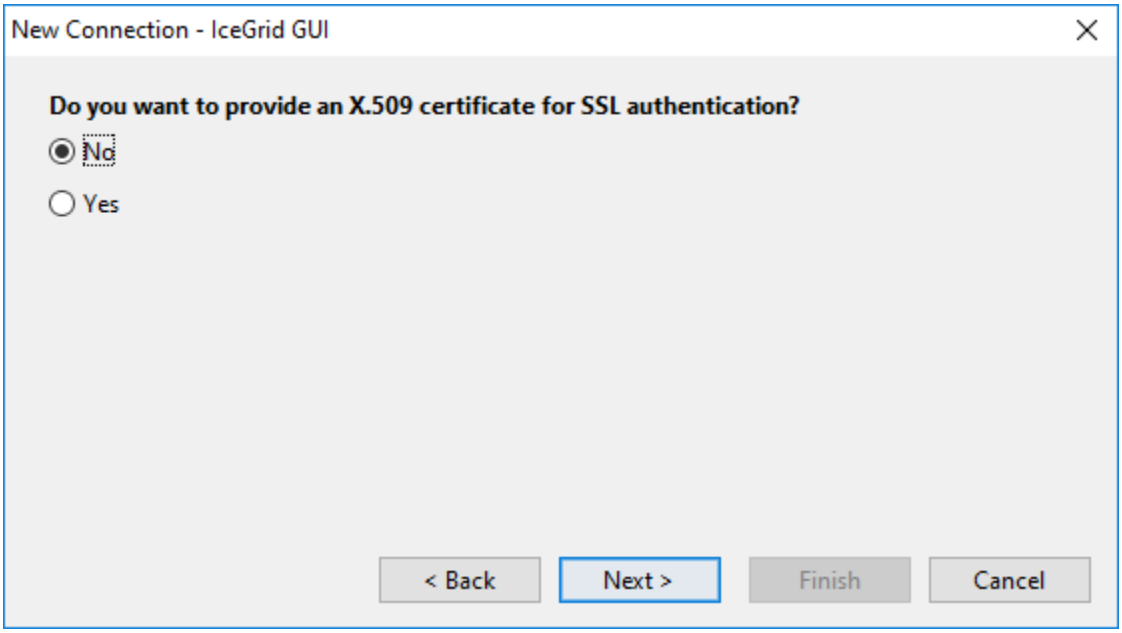
Step 2: Enter the addressing information for the target Glacier2 router as a hostname and port number:



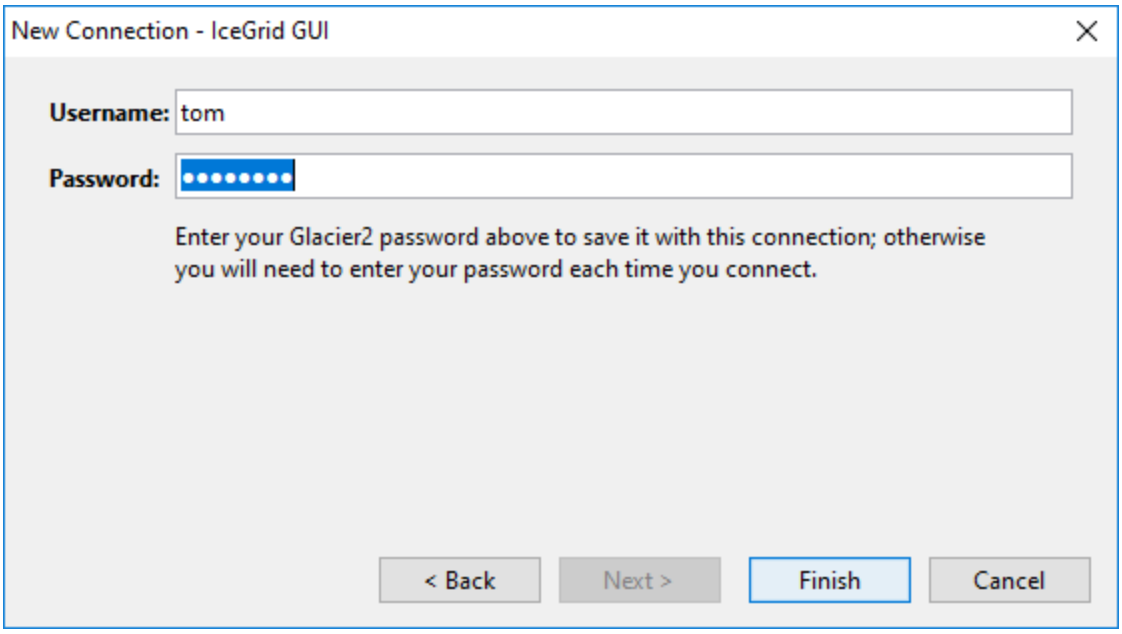
Step 3: Enter the hostname of the Glacier2 router, leave the port number empty to use the default Glacier2 port number (4063 for TCP and 4064 for SSL), and select SSL for the protocol:



Step 4: Choose not to authenticate ourselves for SSL, so we do not provide a X.509 certificate:



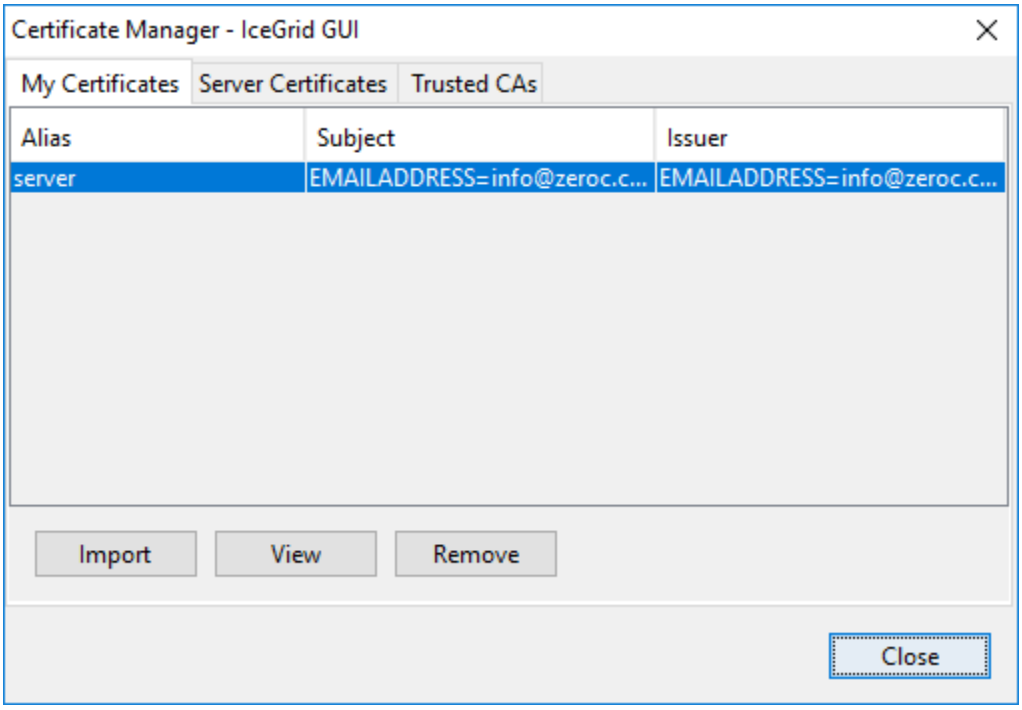
Step 5: Provide a username and password for the connection to the Glacier2 router:



Step 6: Click `Finish` to save the connection; IceGrid GUI then attempts to connect to the IceGrid registry through the Glacier2 router.

SSL Connections and Certificates

IceGrid GUI maintains a persistent store of X.509 certificates for SSL connections with IceGrid registries. You can import, view and remove these certificates with the Certificate Manager. Use `File > Certificate Manager...` or click on the `Import...` button in the Connection wizard to open the Certificate Manager:



The Certificate Manager maintains three sets of certificates:

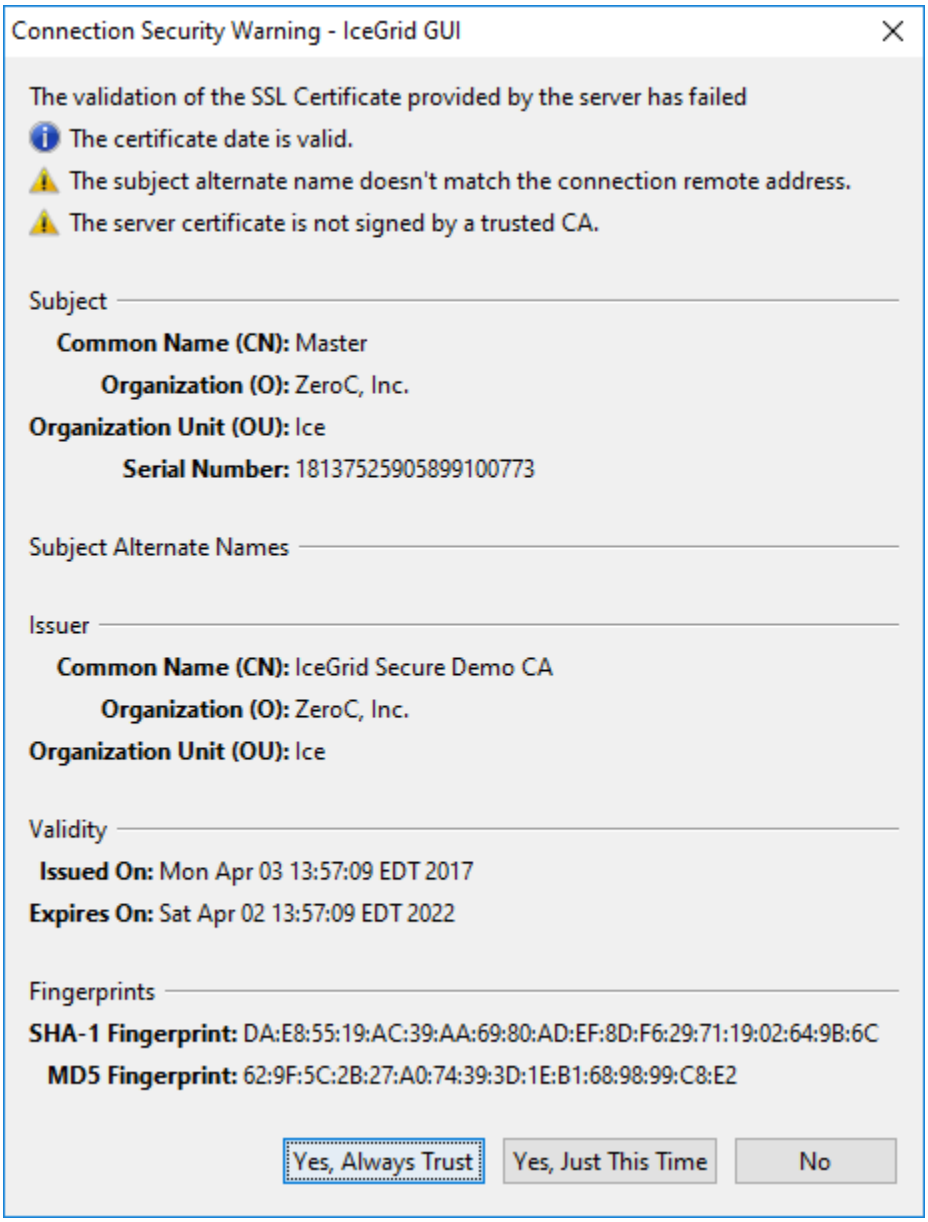
- **My Certificates**
  These certificates are used to authenticate IceGrid GUI when establishing an SSL connection with an IceGrid registry or Glacier2 router (this first authentication is at the SSL level); once the SSL connection is established, this certificate can also be used to authenticate with the target IceGrid registry or Glacier router (at the application level).

- **Server Certificates**
  The certificates of IceGrid registries and Glacier2 routers that IceGrid GUI trusts when establishing an SSL connection. All certificates signed by a Trusted CA (see below) are automatically trusted and usually do not need to be imported in this set.

- **Trusted CAs**
  The certificates of Certificate Authorities.

You do not need to import server certificates or CA certificates prior to establishing an SSL connection with an IceGrid registry or Glacier2 router. IceGrid GUI performs the following checks when establishing an SSL connection:

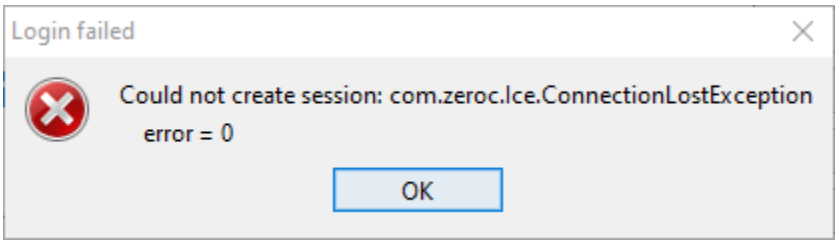
- If the X.509 certificate presented by the IceGrid registry or Glacier2 router matches a Server Certificate, proceed
- Otherwise, if this certificate is signed by a trusted CA, is valid (*now* is within the certificate's validity period) and its alternate name is a match for the SSL connection remote address, proceed
- Otherwise, display a Connection Security Warning dialog similar to the dialog below to let you decide whether or not to proceed with this certificate:



If you select `Yes, Always Trust`, the certificate is added in the persistent Server Certificates set.

A "client" X.509 certificate (saved in My Certificates) is only necessary when the target IceGrid registry or Glacier2 router requires one. This depends on the setting of the `IceSSL.VerifyPeer` property in those servers: when `IceSSL.VerifyPeer` is 2, IceGrid GUI must provide

a valid certificate. If you forget to provide a certificate, or provide an invalid certificate, the connection establishment will fail and you will get an error dialog such as:



Editing a Saved Connection

In the `Saved Connections` dialog, click on `Edit Connection` to edit a connection. This opens the Connection wizard for your saved connection. With this wizard, IceGrid GUI does not attempt to connect to the target IceGrid registry when you click on the `Finish` button.

Closing a Connection

Use `File > Logout` or press the

button to disconnect from an IceGrid registry. This clears all information in the Live Deployment pane.

Run Time Components

IceGrid GUI shows the following run time components:

- Registry ()
  Represents the IceGrid registry process with which IceGrid GUI communicates
- Slave Registry ()
  Represents a slave registry process, when you have several replicated IceGrid registries
- Node ()
  Represents an IceGrid node process
- Regular Server ()
  Represents a regular server process, started and monitored by an IceGrid node
- IceBox Server ()
  Represents an IceBox server process, started and monitored by an IceGrid node
- Adapter ()
  Represents an Ice indirect object adapter, deployed within a server or a service
- Database Environment ()
  Represents a Freeze/Berkeley DB Database Environment, deployed within a server or a service
- IceBox Service ()
  Represents an IceBox service, deployed on an IceBox server
- Metrics View ()
  Represents a Metrics view associated with an Ice server, an Ice service, an IceGrid node, the Registry or a Slave Registry

All actions on these components (through menus, buttons and keyboard short-cuts) affect directly and immediately the running IceGrid deployment.

Registry Run Time Component

The registry is the root node of the Run Time Components tree, and represents the IceGrid registry process administered by IceGrid GUI.

On this page:

Actions

A registry provides the following actions, from its contextual menu and from the `Tools > Registry` menu:

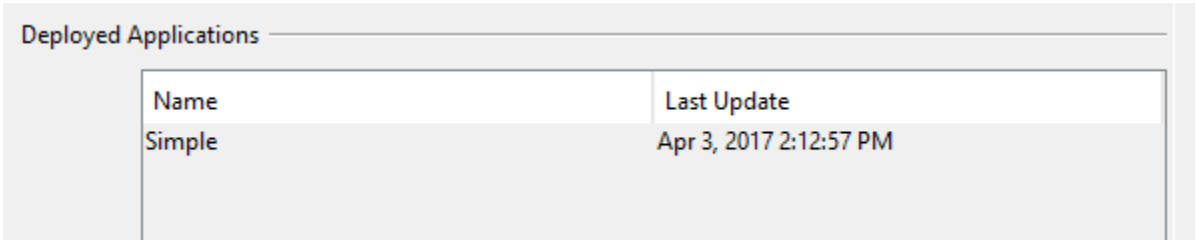
- **Add Well-Known Object**
  Create a new dynamic well-known object in the IceGrid registry.
- **Retrieve Ice log**
  Retrieve the log messages sent to the IceGrid registry's logger into an Ice Log Dialog. The Ice Log Dialog attaches a remote logger t o the registry's logger.
- **Retrieve stdout**
  Retrieve the IceGrid registry's stdout into a Log File Dialog. This retrieval succeeds only when the registry's stdout output has been redirected to a file using the `Ice.StdOut` property.
- **Retrieve stderr**
  Retrieve the IceGrid registry's stderr into a Log File Dialog. This retrieval succeeds only when the registry's stderr output has been redirected to a file using the `Ice.StdErr` property.
- **Shutdown**
  Shutdown the registry process.

> You cannot restart an IceGrid registry from IceGrid GUI.

Properties

The Registry Properties panel shows:

- **Hostname**
  The name of the host on which the IceGrid registry process is running.
- **Build Id**
  The build Id of this registry: this corresponds to the Ice property BuildId.
- **Properties**
  A table showing all the Ice properties currently set in this registry.
- **Deployed Applications**

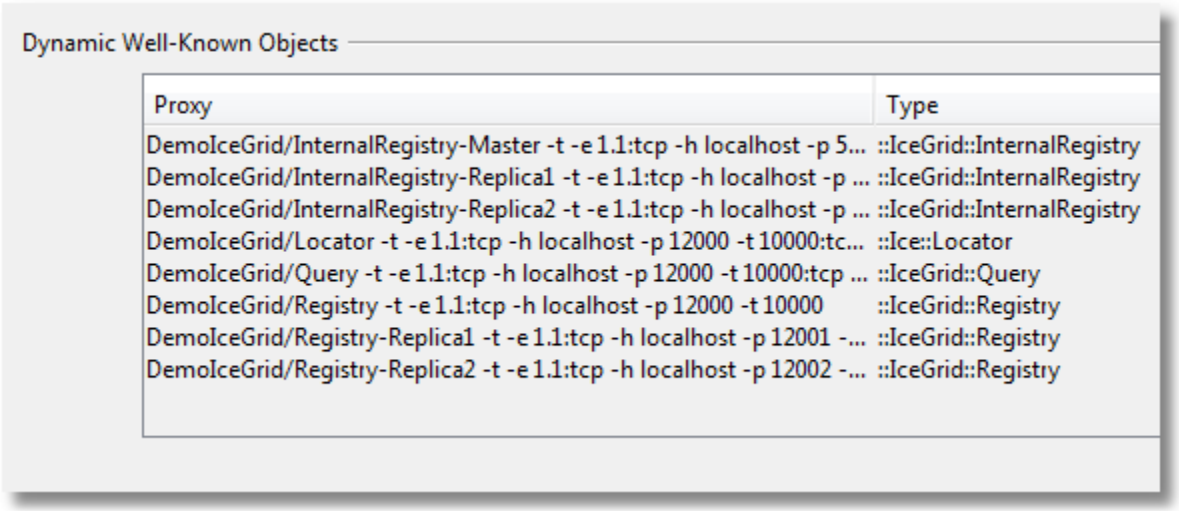


This table shows all the applications deployed on this IceGrid registry, along with the date and time of the last update of each application. A contextual menu allows you to:

- open the corresponding application descriptor
- patch the application, that is, instruct the IceGrid nodes to download the latest version of this application's files
- show additional details on this application in this registry

- remove (undeploy) the application from the registry

- **Dynamic Well-Known Objects**



This table shows the well-known objects registry dynamically with the IceGrid registry: well-known objects defined using adapter and replica-group definitions are not included. A contextual menu allows you to add or remove entries from this table, and to show a given entry in its own dialog.

- **Dynamic Object Adapters**

This table shows the object adapters registered dynamically with the registry. It is typically empty. A registry allows dynamically registered adapters only when its `IceGrid.Registry.DynamicRegistration` property is set to a value greater than 0. A contextual menu allows you to remove entries from this table.

Note that application filtering does not affect this panel: all applications, well-known objects and dynamic object adapters are always displayed.

Children

A registry node can have the following types of children:

- Metrics View
- Slave Registry
- Node

Slave Registry Run Time Component

An IceGrid deployment may use several IceGrid registry replicas, with one Master registry and a number of read-only Slave registries.

Actions

A slave registry provides the following actions, from its contextual menu and from the `Tools > Registry` menu

- **Retrieve Ice log**
  Retrieve the log messages sent to the IceGrid registry's logger into an Ice Log Dialog. The Ice Log Dialog attaches a remote logger t o the registry's logger.
- **Retrieve stdout**
  Retrieve the IceGrid registry's stdout into a Log File Dialog. This retrieval succeeds only when the registry's stdout output has been redirected to a file using the `Ice.StdOut` property.
- **Retrieve stderr**
  Retrieve the IceGrid registry's stderr into a Log File Dialog. This retrieval succeeds only when the registry's stderr output has been redirected to a file using the `Ice.StdErr` property.
- **Shutdown**
  Shutdown the registry process.

> You cannot restart an IceGrid registry from IceGrid Admin.

Properties

The Registry Properties panel shows:

- **Hostname**
  The name of the host on which the IceGrid registry process is running.
- **Build Id**
  The build Id of this Registry: this corresponds to the Ice property BuildId.
- **Properties**
  A table showing all the Ice properties currently set in this IceGrid registry.

Children

A slave registry can have the following types of children:

- Metrics View

Node Run Time Component

A node represents an IceGrid node process registered with the IceGrid registry.

On this page:

States

A node can be either up (

) or down (

). A "down" node is shown only when it is described by an application deployed on this IceGrid registry.

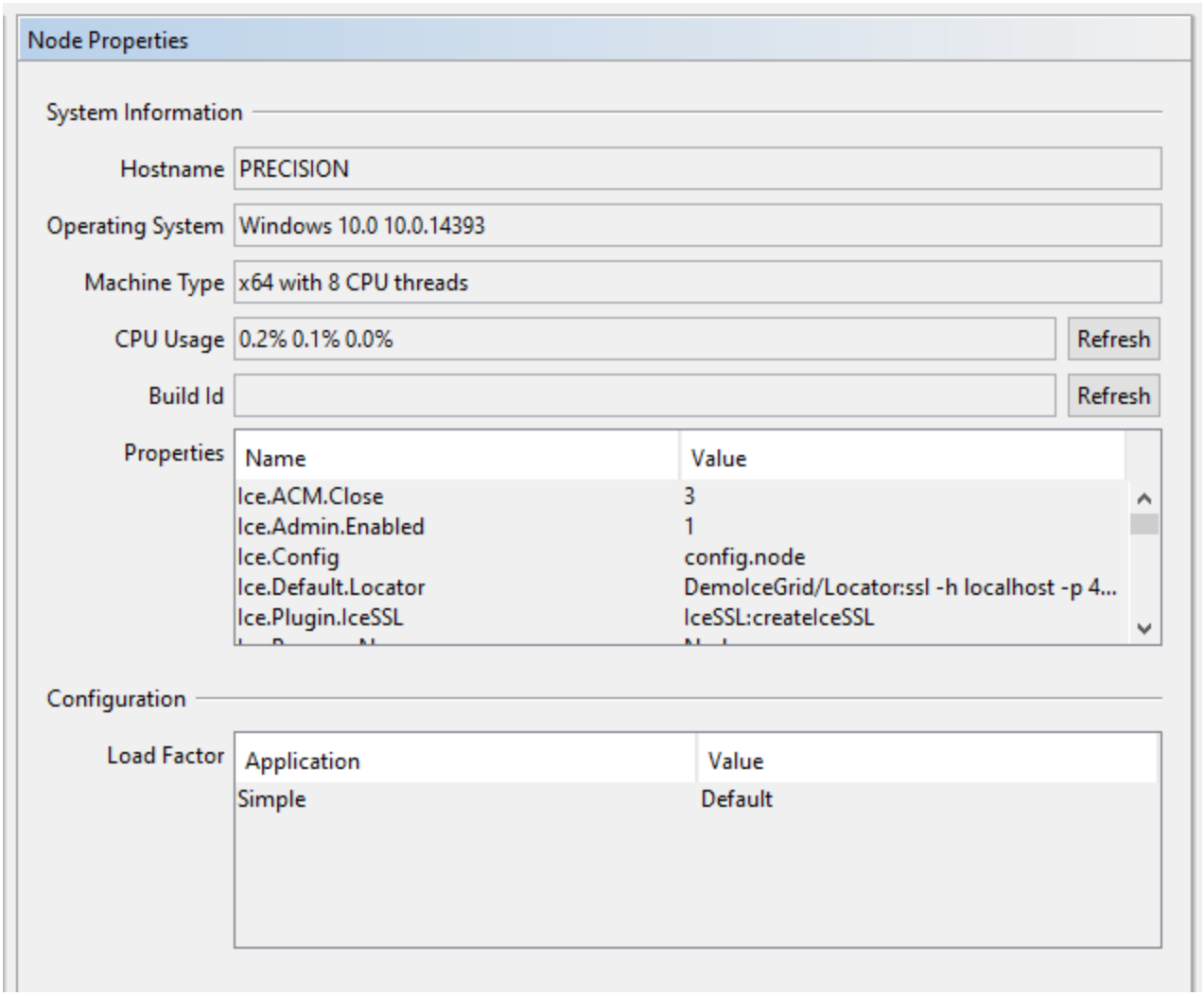
Actions

A node provides the following actions, from its contextual menu and from the `Tools > Node` menu:

- **Retrieve Ice log**
  Retrieve the log messages sent to the IceGrid node's logger into an Ice Log Dialog. The Ice Log Dialog attaches a remote logger to the node's logger.
- **Retrieve stdout**
  Retrieve the IceGrid node's stdout into a Log File Dialog. This retrieval succeeds only when the node's stdout output has been redirected to a file using the `Ice.StdOut` property.
- **Retrieve stderr**
  Retrieve the IceGrid node's stderr into a Log File Dialog. This retrieval succeeds only when the node's stderr output has been redirected to a file using the `Ice.StdErr` property
- **Shutdown**
  Shutdown the IceGrid node process.

> You cannot restart an IceGrid node from IceGrid GUI.

Properties



The Node Properties panel shows:

- **Hostname**
  The name of the host on which the IceGrid node process is running.
- **Operating System**
  The operating system name and version of the host on which the IceGrid node process is running.
- **Machine Type**
  The type of CPUs and the number of CPU threads of the host on which the IceGrid node process is running. For example, a computer with a dual-core CPU and 2 "hyper threads" per core will show 4 CPU threads.
- **CPU Usage (Windows) or Load Average (Linux/Unix)**
  On Windows, shows the percentage of CPU utilization in the past 1, 5, and 15 minutes. On Linux/Unix, shows the load-average in the past 1, 5 and 15 minutes. These values are retrieved when the Node Properties panel is displayed. Click on the Refresh button to retrieve the latest values.
- **Build Id**
  The build Id of this node: this corresponds to the Ice property BuildId.
- **Properties**
  A table showing all the Ice properties currently set in this IceGrid node.
- **Load Factor**
  Shows the load factor defined by each application using this node.

Children

An IceGrid node can only have the following types of children:

- Metrics View
- Regular Server
- IceBox Server

Server Run Time Component

A server represents an Ice server process. It can be either regular server (with typically a single Ice communicator) or an IceBox server hosting a number of IceBox services.

On this page:

States

A server is always in one of the following states (the first icon is for regular servers, the second for IceBox servers):

- Unknown (
  
  ): this state is shown when the parent IceGrid node is down.
- Inactive (
  
  ): the server is not running.
- Activating (
  
  ): the server is starting up. The IceGrid registry is waiting for the server to register all its object adapters with server lifetime.
- Active (
  
  ): the server is running, and has registered all its object adapters with server lifetime with the IceGrid registry.
- Deactivating (
  
  ): the server is shutting down. The IceGrid registry is waiting for the server process to exit.
- Destroyed (
  
  ): the server being removed of the IceGrid registry. This is a very transient state.

A server can also be either enabled or disabled; when disabled, the icons above are grayed-out. A disabled server cannot be started until it is re-enabled.

Actions

A server provides the following actions, from its contextual menu, from the `Tools > Server` menu, and from buttons on the Server Properties panel:

- **Start**
  Instruct the IceGrid node to start the server.

- **Stop**
  Instruct the IceGrid node to shutdown the server.

- **Enable**
  Mark the server as "enabled".

- **Disable**
  Mark the server as "disabled". A disabled server cannot be started; however an already running server can be marked "disabled".

- **Patch Distribution**
  Patch the server's distribution, that is, instruct the IceGrid node to download the latest files from the server's distribution.

- **Write Message**
  Open a dialog that allows you to write a message to the server's stdout or stderr.

- **Retrieve Ice log**
  Retrieve the log messages sent to the server's logger into an Ice Log Dialog. The Ice Log Dialog attaches a remote logger to the server's logger.

- **Retrieve stdout**
  Retrieve the stdout log file of this server into a Log File Dialog. This retrieval succeeds only when the server's stdout output has been redirected to a file using the `Ice.StdOut` property. This is usually achieved by setting the IceGrid.Node.Output property in the IceGrid node configuration file.

- **Retrieve stderr**
  Retrieve the stderr log file of this server into a Log File Dialog. This retrieval succeeds only when the server's stderr output has been redirected to a file using the `Ice.StdErr` property. This is usually achieved by setting the IceGrid.Node.Output property in the IceGrid node configuration file.

- **Retrieve log file**
  Retrieve a log file of this server into a Log File Dialog.

- **Send Signal**
  Send a signal to a server, for example SIGQUIT. Available only for non-Windows servers.

Properties

The Server Properties panel shows first the Runtime Status of the server, i.e. "live" values retrieved from the server:

- **State**
  The state of the server (Active, Deactivating, Inactive etc., see above)

- **Enabled**
  A checkbox that is checked when the server is enabled.

- **Process Id**
  The process ID of the server.

- **Build Id**
  The build Id of this server: this corresponds to the Ice property BuildId.

- **Properties**
  A table showing all the Ice properties currently set in this server. These properties are retrieved each time you select a new server in IceGrid GUI, and each time you click on the Refresh button next to the Build Id field.

The remaining Server Properties under Configuration come from the IceGrid descriptors associated with this server:

- **Application**

  The name of the application containing this server's definition. The button on the right shows the server definition in an Application tab.

- **Description**
  A free-text description of this server.

- **Properties**
  A table showing all the Ice properties of this server. These properties may come from template definitions, property sets, server-instance properties etc. They are all combined in this table.

- **Path to Executable**
  The path to the server's executable, used by the IceGrid node to start the server.

- **Ice Version**

The Ice version of this server.

- **Working Directory**
  The path to the server's working directory used by the IceGrid node when starting the server.

- **Command Arguments**
  The command-line arguments given to the server when started by IceGrid.

- **Run as**
  On Linux/Unix, a server may be started as a specific user when IceGrid node runs as root. Run as shows this username. When blank, the server runs as the IceGrid node user except if IceGrid node runs as root (on Linux/Unix); in this case, the server runs as nobody.

- **Environment Variables**
  A server started by IceGrid node gets these environment variables in addition to the environment variables inherited from the IceGrid node.

- **Activation Mode**
  The server's activation mode.

- **Activation Timeout**
  The server's activation timeout.

- **Deactivation Timeout**
  The server's deactivation timeout.

- **Allocatable**
  This checkbox Shows whether this server is allocatable or not.

- **Depends on the application distribution**
  If this checkbox is checked, this server depends on the enclosing application's distribution: each time this distribution is patched, the server is automatically shut down before the patch.

- **IcePatch2 Proxy**
  A stringified or well-known proxy for the IcePatch2 server than contains this server's distribution. When blank, this server does not have its own distribution.

- **Directories**
  List of directories included in the server distribution. When blank, the entire IcePatch2 server repository is used as the server distribution.

Children

A regular server node can have the following types of children:

- Metrics View
- Adapter
- Database Environment

An IceBox server node can have the following types of children:

- Metrics View
- Service

Adapter Run Time Component

An adapter represents an Ice object adapter described in the IceGrid registry that resides in a server or an IceBox service. Note that direct object adapters are not displayed since IceGrid knows nothing about them.

On this page:

States

An adapter can be either active (

) or inactive (

).

Properties

The Adapter Properties panel shows first the Runtime Status of the object adapter, i.e. "live" values retrieved from the enclosing server or service:

- **Status**
  The status of this object adapter: Active or Inactive.

- **Published Endpoints**
  The endpoints that this object adapter has registered with the IceGrid registry. Blank when the adapter's status is Inactive.

The remaining Server Properties under Configuration come from the IceGrid descriptor that defined this object adapter:

- **Description**
  A free-text description of this adapter.

- **Adapter ID**
  This object adapter's ID. Adapter IDs are unique within an IceGrid deployment.

- **Replica Group**
  When this adapter belongs to a replica group, shows the replica group ID. Blank otherwise.

- **Priority**
  The adapter's priority. Used only when the adapter belongs to a replica group with the "Ordered" load balancing policy.

- **Endpoints**
  The Endpoints configuration property of this object adapter.

- **Published Endpoints**
  The PublishedEndpoints configuration property of this object adapter.

- **Register Process**
  When the Register Process checkbox is checked, this object adapter will register an `Ice::Process` object with IceGrid upon activation. This setting is meaningful only for servers with an Ice version less than 3.3.

- **Server Lifetime**
  When the Server Lifetime checkbox is checked, IceGrid considers that this object adapter is activated when the server starts up and deactivated when the server shuts down. This allows IceGrid to detect when a server is fully activated (all its object adapters with a server-lifetime have registered) and when a server is shutting down (at least one object adapter with server-lifetime has unregistered).

- **Well-known Objects**
  The well-known objects defined by this object adapter.

- **Allocatable Objects**
  The allocatable objects defined by this object adapter.

Database Environment Run Time Component

A Database Environment (

) represents a Berkeley DB database environment, used by the Freeze persistence service.

Properties

The Database Environment Properties panel shows:

- **Description**
  A free-text description of this Berkeley DB database environment.
- **DB Home**
  The path to the home directory of this database environment, which corresponds to the Freeze property `Freeze.DbEnv.`*`env-name`*`.DbHome`. It is often created by the IceGrid node in the server's private directory.
- **Properties**
  Berkeley DB configuration properties for this database environment. IceGrid generates a `DB_CONFIG` file in the DB Home directory with these properties.

Service Run Time Component

A service represents an IceBox service loaded (or potentially loaded) within an IceBox server.

On this page:

States

A service can be either started (

) or stopped (

) within an IceBox server.

Actions

An IceBox service provides the following actions, from its contextual menu, from the `Tools > Service` menu, and from buttons on the Service Properties panel:

- **Start**
  Instruct the IceBox server to start the service.
- **Stop**
  Instruct the IceBox server to stop this service.
- **Retrieve Ice log**
  Retrieve the log messages sent to the service's logger into an Ice Log Dialog. The Ice Log Dialog attaches a remote logger to the service's logger.
- **Retrieve log file**
  Retrieve the log file of this service into a Log File Dialog.

Properties

The Service Properties panel shows first the Runtime Status of the service, i.e. "live" values retrieved directly from the service:

- **State**
  A checkbox that is checked when the service is started.
- **Build Id**
  The build Id of this service: this corresponds to the Ice property `BuildId`.
- **Properties**
  A table showing all the Ice properties currently set in this service. These properties are retrieved each time you select a new service in IceGrid GUI, and each time you click on the Refresh button next to the Build Id field.

The remaining Server Properties come from the IceGrid descriptors associated with this service:

- **Description**
  A free-text description of this service.
- **Properties**
  A table showing all the Ice properties of this service. These properties may come from template definitions, property sets, service-instance properties etc. They are all combined in this table.
- **Entry Point**
  The entry point for this service. This corresponds to the value of the `IceBox.Service.`*`name`* property.

Children

An IceBox service can have the following types of children:

- Metrics View
- Adapter
- Database Environment

Metrics View Run Time Component

A Metrics View displays the Metrics maps associated with a Server or Service.

On this page:

States

A metrics view can be either enabled (

) or disabled(

). Once enabled, a Metrics View may degrade the performance of the instrumented server or service.

Actions

An Metrics View provides the following actions, from its contextual menu and from the `Tools > Metrics View` menu:

- **Enable**
  Enable this Metrics View.

- **Disable**
  Disable this Metrics View.

Metrics Report

The Metrics Report panel shows the maps included in the Metrics View. The columns of the maps include the metrics themselves (for example, the total number of operations dispatched by the server since the Metrics view was enabled) and computed values (for example, the average lifetime of an operation dispatch, since the Metrics view was enabled).

> Tool tips on each column describe the metrics or computed value displayed by the column.

IceGrid GUI retrieves the latest metrics every 5 seconds and automatically refreshes the current Metrics Report.

Note that IceGrid GUI displays only non-empty metrics maps; for example, if a server does not make any remote invocation, its Invocation map is empty and will not be displayed.

Application Component

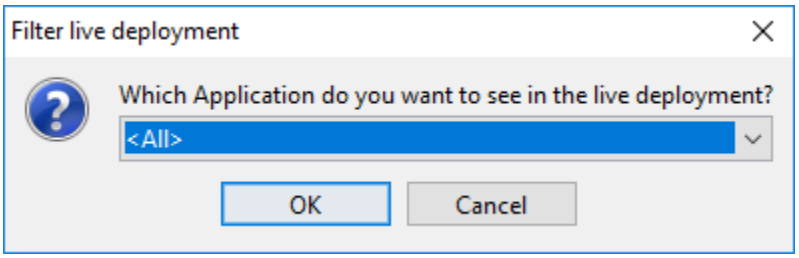
An application is primarily a way to group definitions and does not actually exist in a live deployment.

Nevertheless, IceGrid GUI offers a number of application-related actions in the live deployment view through the Filter live deployment button and the `Tools > Application` menu:

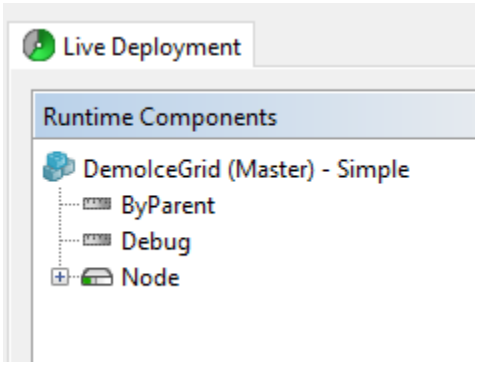
- **Patch Distribution**
  Instruct the IceGrid nodes to download the latest version of the files associated with the selected application.

- **Show details**
  Show various information about the selected application.

- **Filter live deployment**
  Shows only the selected application in the live deployment view; servers and other components associated with other applications are "filtered out".



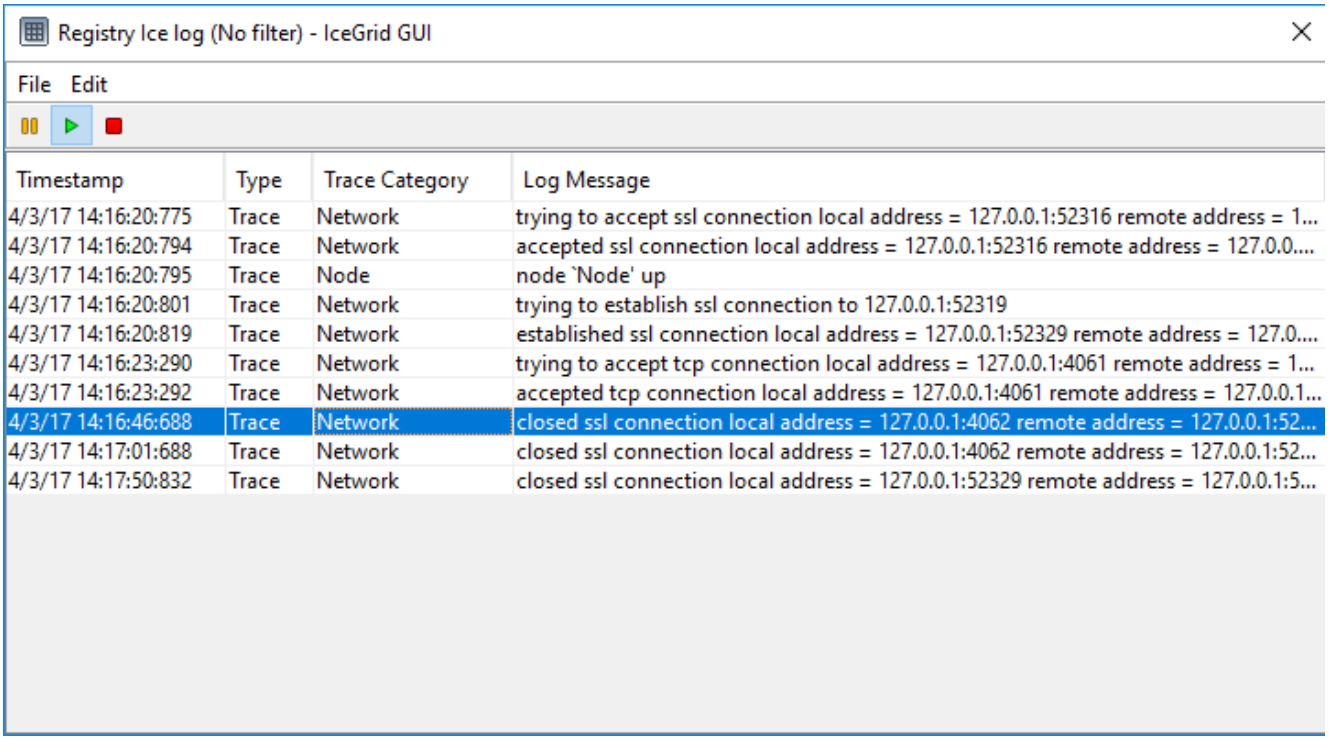
When filtering is enabled, the name of the selected application appears to the right of the registry name. In the picture below, the live deployment view shows only the application named 'Simple'.



- **Remove from registry**
  Remove the selected application from the registry.

Ice Log Dialog

The Ice Log dialog shows the Ice log messages of a server, service, node or registry retrieved through IceGrid. The monitored Ice application sends these log messages to a remote logger implemented by the Ice Log dialog.



On this page:

States

An Ice log dialog is always in one of the following states:

- **Running**
  The dialog is attached to the monitored application's logger: it displays new log messages as soon as it receives them.

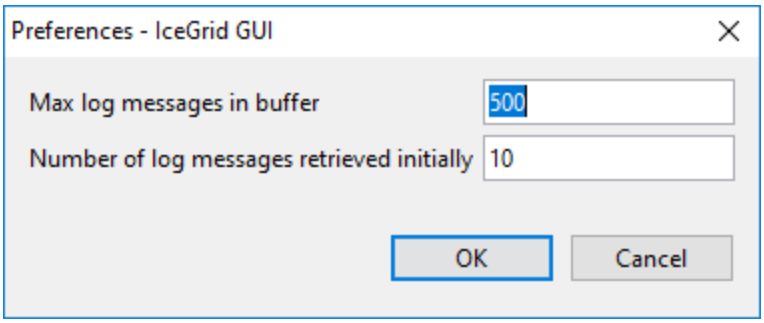
- **Paused**
  The dialog is attached to the monitored application's logger: it queues new log messages, but does not display them.

- **Stopped**
  The dialog is no longer attached to the monitored application's logger.

When an Ice log dialog is running or paused, and the target server, node or registry is stopped, the Ice log dialog automatically transitions to the Stopped state. However, when an Ice log dialog is attached to an IceBox service, the Ice log dialog's state does not change when only the service is stopped, because stopping a service does not shutdown or destroy the associated communicator.

Preferences

Use the `Edit > Preferences...` menu to open the Preferences dialog. These preferences apply to the current dialog and to any Ice Log dialog opened later on.

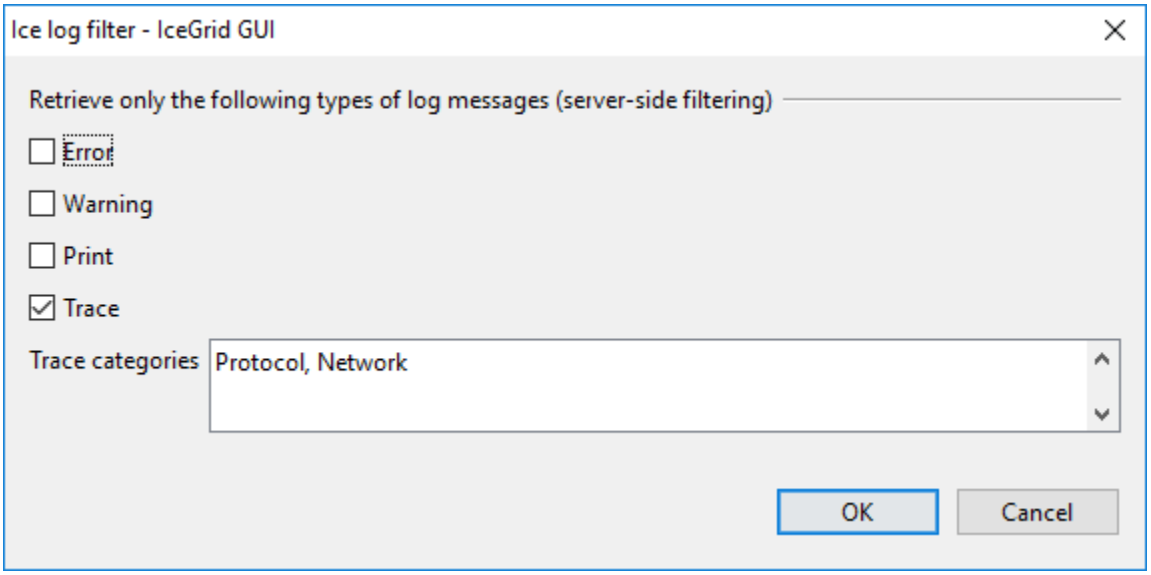


This dialog allows you to view and update the following settings:

- **Max log messages in buffer**
  The maximum number of lines displayed in the Ice log dialog.
- **Number of log messages retrieved initially**
  When a new dialog is opened, or when restarting a stopped dialog, the dialog retrieves and displays up to this number of lines.

Filter

Use the `Edit > Filter...` menu to open the Filter dialog. This filter applies only to the current Ice Log dialog.

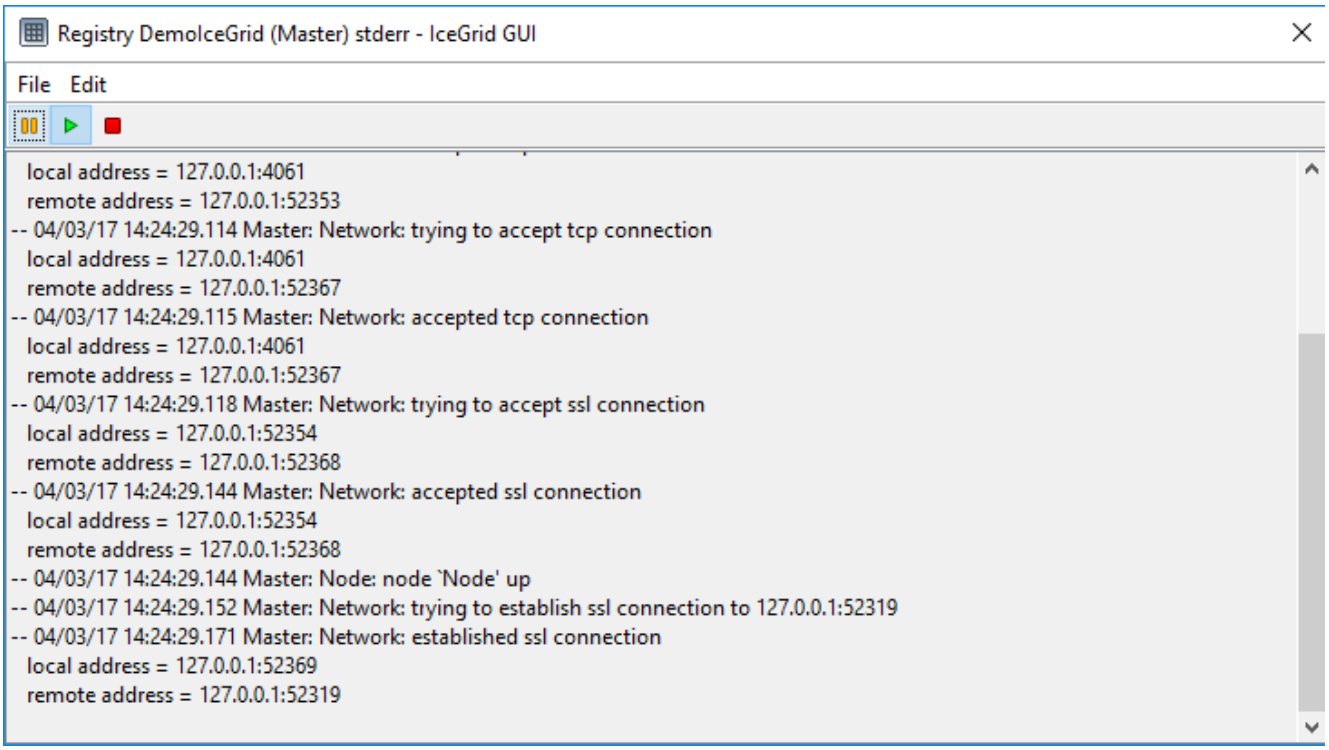


This dialog allows you to filter the log messages sent by the monitored Ice applications. All the filtering is done in this remote Ice application.

- **Error, Warning, Print and Trace Check Boxes**
  Check the type of log messages you want to receive.
- **Trace Categories**
  Enter the list of trace categories you want to receive, or leave empty to receive trace messages with any trace category.

Log File Dialog

The Log File dialog shows a log file (text file) retrieved through IceGrid. Often, a process will be writing to this log file and the dialog will periodically check for new lines to display.



On this page:

States

A log file dialog is always in one of the following states:

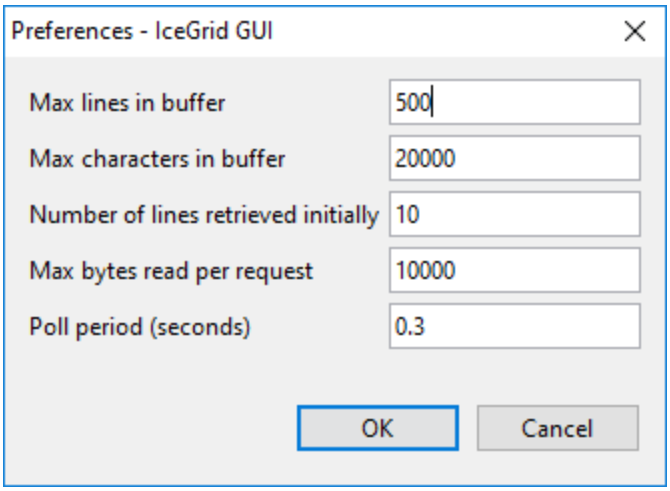
- **Running**
  The dialog is periodically checking for new lines.

- **Paused**
  The dialog is "connected" to a remote log file but does not currently retrieve new lines.

- **Stopped**
  The dialog is not retrieving new lines from the remote log file.

Preferences

Use the `Edit > Preferences...` menu to open the Preferences dialog. These preferences apply to the current dialog and to any Log File dialog opened later on.

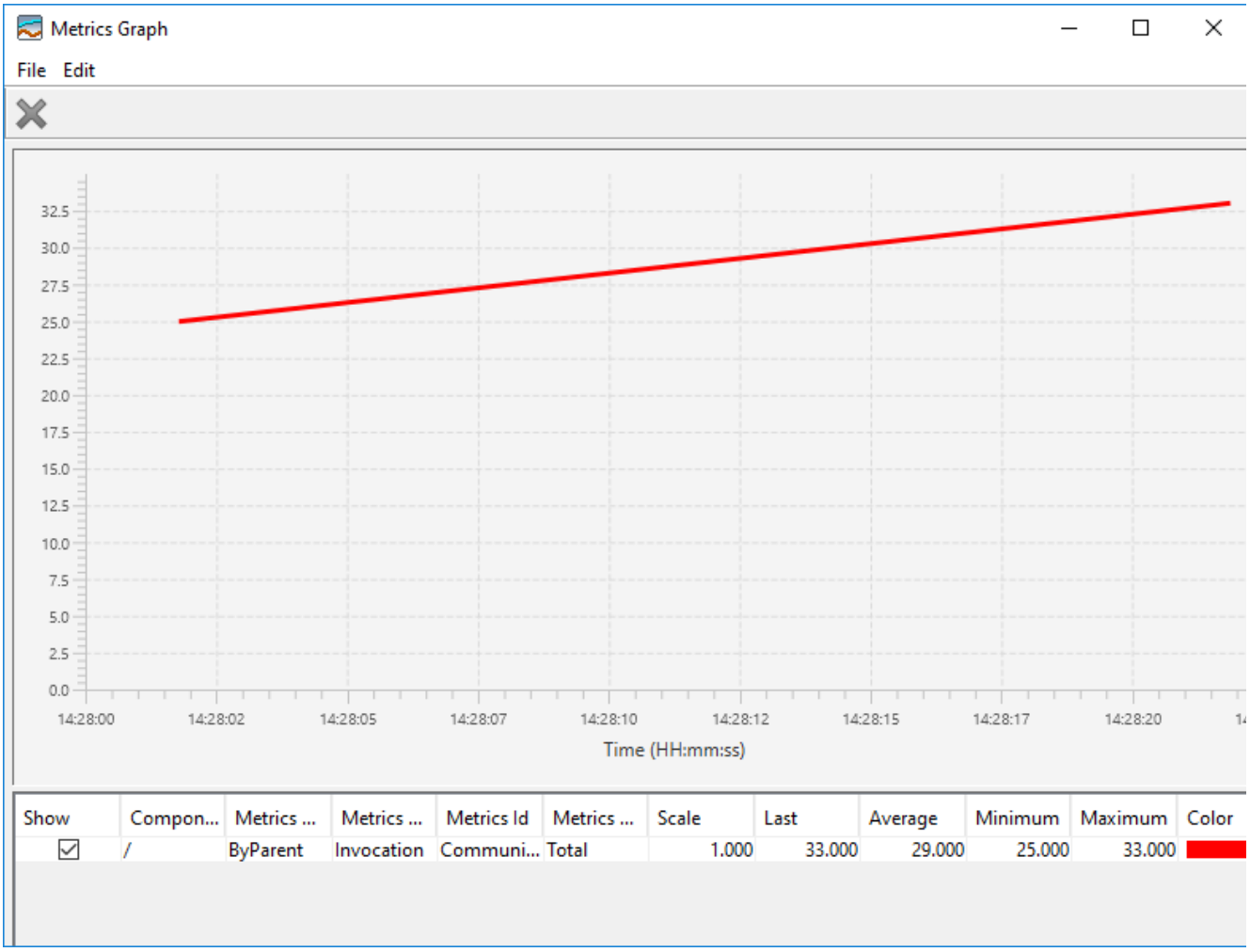


This dialog allows you to view and update the following settings:

- **Max lines in buffer**
  The maximum number of lines displayed in the log dialog.
- **Max characters in buffer**
  The maximum number of characters displayed in the log dialog.
- **Number of lines retrieved initially**
  When a new dialog is opened, or when restarting a stopped dialog, the dialog retrieves and displays up to this number of lines.
- **Max bytes read per request**
  The maximum number of bytes retrieve by each request. Pick a value that is low enough to make the dialog appear responsive and big enough to avoid many round-trips when lots of data are logged. IceGrid GUI requires a value between 100 and Ice.MessageSize Max - 512.
- **Poll period**
  When in the running state, the dialog attempts to retrieve new lines from the log file (through IceGrid) every Poll period seconds.

Metrics Graph

A Metrics Graph displays metrics retrieved from one or more Ice servers or IceBox services.



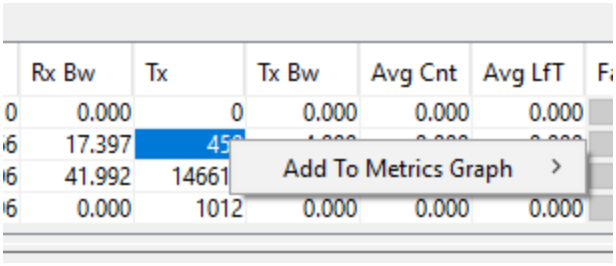
The horizontal axis corresponds to the time, while the vertical axis plots the corresponding metrics values.

> The Metrics Graph feature requires Java 8 with JavaFX.

On this page:

Creating a New Metrics Graph

You can create a new Metrics Graph window with the `File > New > Metrics Graph` menu, or with a contextual menu over a metric in a Metrics View, for example:



Adding and Removing Metrics

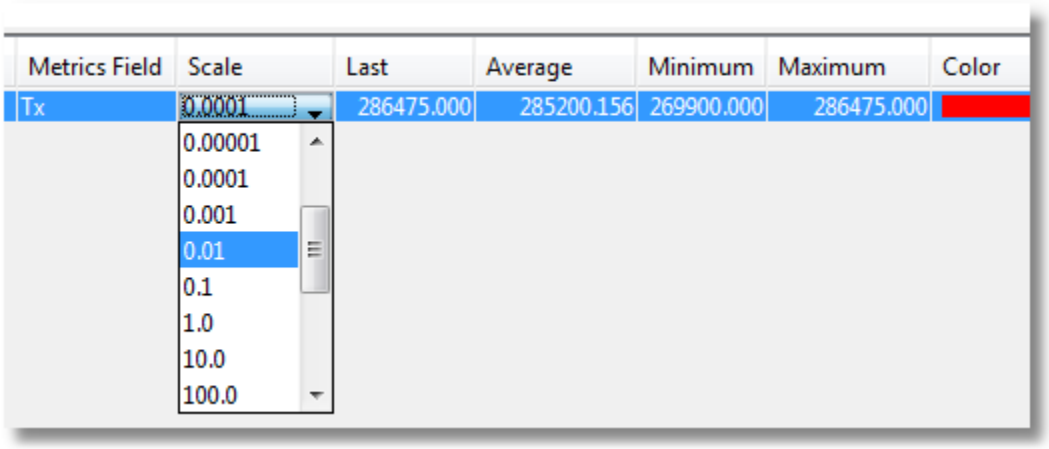
A metric is added to a Metrics Graph with drag & drop, or through a contextual menu over this metric.

To remove a metric from a Metrics Graph, select this metric and press Delete or the

button.

Adjusting Scale and Color

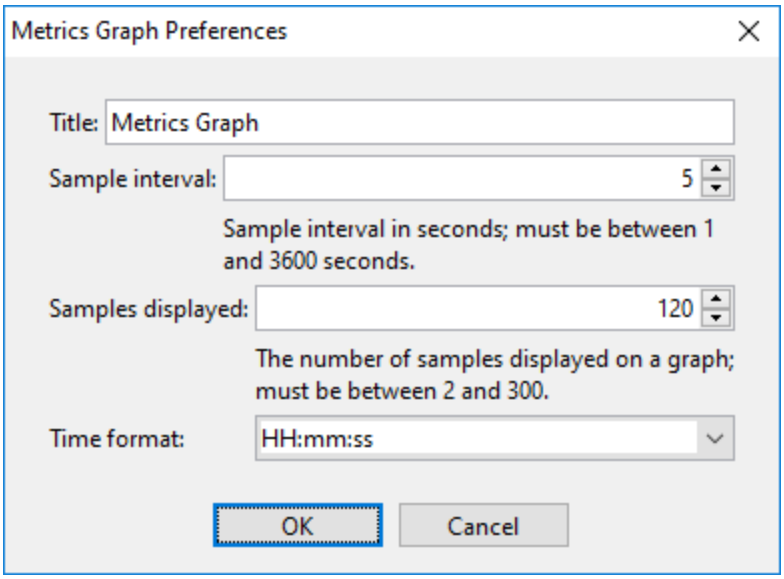
You can display metrics with different scales on the same graph by adjusting the scale factor of each metric. The values plotted are the metric's values times the scale factor; the default scale factor is 1.0.



You can also change the color used to display a metric's line by clicking on the Color cell of the metric.

Metrics Graph Preferences

The `File > Preferences` menu of a Metrics Graph window opens a dialog that allows you to configure several properties of your Metrics Graph:



For each metric plotted in a Metrics Graph, IceGrid GUI retrieves the corresponding value every `Sample interval` seconds (by default every 5 seconds), and displays n values (n = 120 by default). `Sample interval` times `Samples displayed` correspond to the time period represented on a graph; by default, it is 5s * 120 = 10 minutes.

Application Tabs

IceGrid definitions are organized in "applications", with typically one or a few applications deployed on a given IceGrid instance.

An Application tab allows you to:

- create a new application: describe servers, IceBox services, the nodes you want to run them, specify load-balancing policies, etc.
- browse and edit the definition of an existing, live application deployed in an IceGrid instance
- browse and edit the definition of an application stored in an IceGrid XML file.

Topics

Editing and Saving IceGrid Descriptors

On this page:

Editing

As soon as you make any update in a form, IceGrid GUI enables two buttons at the bottom of this form: Apply and Discard.

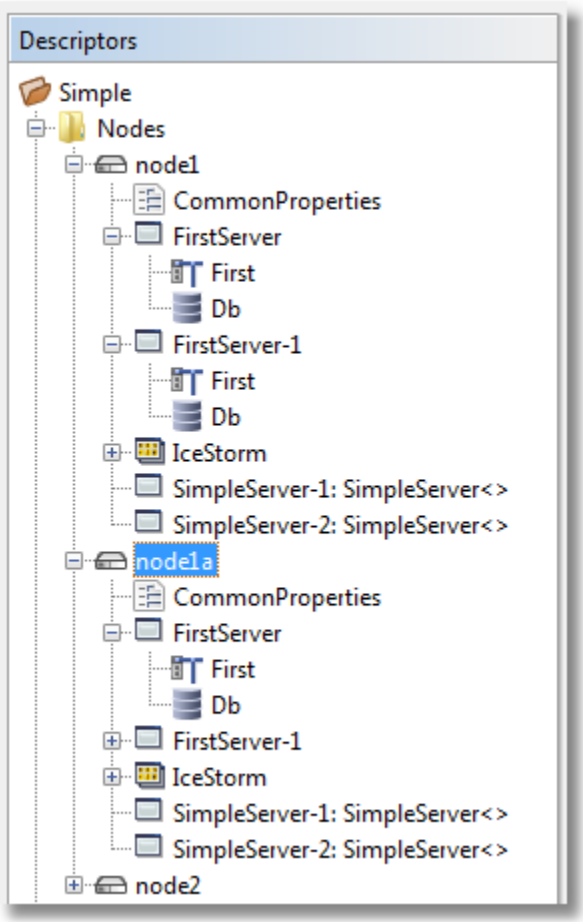
If you navigate to another node without pressing Apply or Discard, the default is Apply: your changes are applied to the in-memory representation of the application definition. However these changes are not stored to the IceGrid registry or XML file until you save the application definition (see below).

Editing a live application (

) also disconnects this application from the IceGrid registry: updates made by other users are no longer propagated to the Application tab.

Copy & Paste

Most descriptor sub-trees can be copied and later pasted. Copies are always deep-copies: for example if you copy a node, all the servers on this code are copied, including all the the sub-elements of these servers (object adapters, database environment, services, etc.).



After pasting a sub-tree, you typically need to check and edit the new elements to avoid any duplicate server IDs, adapter IDs etc.

Error Checking

IceGrid GUI performs very little error checking while you are working on an application definition. For example, you may temporarily keep several servers with the same ID, leave some parameters of a template instance unset, or use an undefined variable. All such errors are only detected when you save your application to an IceGrid registry.

There are nonetheless two types of constraints enforced by IceGrid GUI at all times:

- two descriptor nodes in an application tab display cannot have the same name.
- some fields (such as Path to Executable for a server) cannot be empty.

If you violate such a constraint, IceGrid GUI prevents you from applying your change.

Saving

You save an application definition to an IceGrid registry or an XML file with the menu item `File > Save`, `File > Save to File`, `File > Save to Registry (Servers may restart)`, `File > Save to Registry (No Server restart)` or with the corresponding toolbar buttons.

`Save` is equivalent to `Save to Registry (Servers may restart)` for a live application, and to `Save to File` for a file-bound application.

Saving an application to the IceGrid registry causes the registry to distribute any relevant changes to the affected nodes. In turn, each node applies those changes to its servers. If a server is running at the time of an update, the node may need to stop and restart it, depending on the changes that you make. Consequently, saving to the registry could cause a disruption in service to any clients that are actively using the affected servers.

The following application changes do *not* require a restart:

- Adding, modifying, or removing a server's configuration properties
- Adding new servers

All other changes will require a restart. IceGrid GUI provides two versions of the `Save to Registry` command, one that allows restarts and one that does not. To avoid accidentally causing any disruption in service, we recommend using the `No server restart` option first; this command will fail if any of your updates require a restart. At that point, you can decide whether to force the servers to restart using the other Save command.

> If you change a server's configuration properties with `Save to Registry (No Server restart)`, IceGrid updates the stored properties of this server, and also the properties of your running server instance through its Properties Facet. If these properties are only read by the server at start-up, this may not have the desired effect. If you want to trigger a server restart even when only properties have changed, use `Save to Registry (Servers may restart)`.

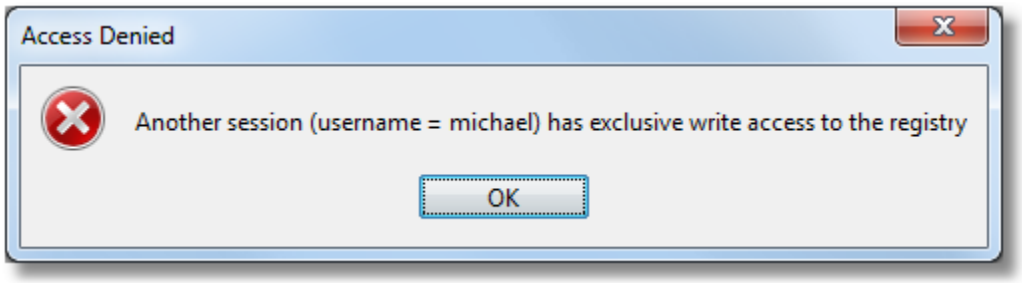
Discarding Updates

You may discard all your updates by selecting `File > Discard Updates` or pressing the corresponding toolbar button. `Discard Updates` simply reloads the application from the IceGrid registry or its associated XML file.

Concurrent Updates to the same IceGrid Registry

If several administrators update the same application definition concurrently, the last save will silently overwrite previous (concurrent) updates.

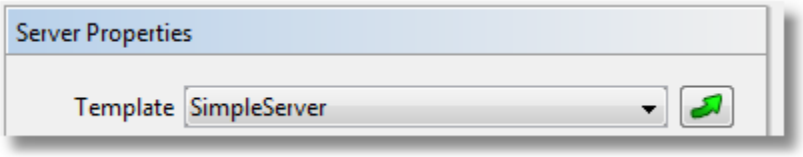
To avoid this situation, you can acquire an exclusive write access to the IceGrid registry with `File > Acquire Exclusive Write Access`. After this exclusive write access is granted, any attempt by another session to save to the IceGrid registry will result in an error:



Navigation within an Application Tab

Each Application tab maintains a history of the nodes you have visited in this tab. You can navigate these nodes using the `View > Go Backto the Previous Node` and `View > Go to the Next Node` menu items, or with the equivalent toolbar buttons:

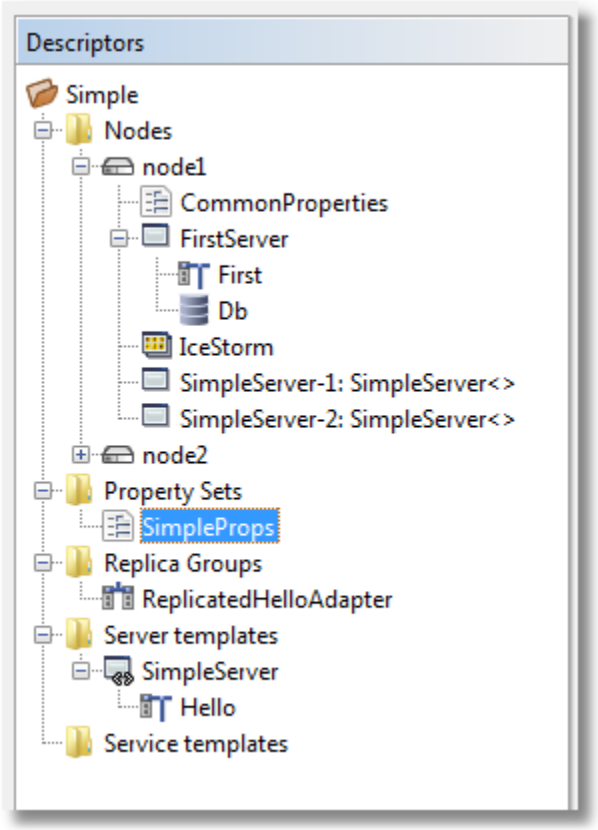
Also on some forms a green arrow button provides a link to another node (typically a template) in the same Application pane:



IceGrid Descriptors

An application definition is described using one application descriptor with a number of nested sub-descriptors (and sub-sub descriptors). The descriptors correspond to Slice types in the IceGrid registry interfaces, and to XML element in an IceGrid XML representation.

IceGrid GUI maps these descriptors to nodes on a tree representation:



Topics

- Variables in IceGrid Descriptors
- Application Descriptor
- Node Descriptor
- Server Descriptor
- Service Descriptor
- Adapter Descriptor
- Database Environment Descriptor
- Property Set Descriptor
- Replica Group Descriptor
- Server Template Descriptor
- Service Template Descriptor

Variables in IceGrid Descriptors

Variables allow you to define commonly-used information once and refer to them symbolically throughout your application descriptors.

On this page:

Syntax

Substitution for a variable or parameter VP is attempted whenever the symbol ${VP} is encountered, subject to the limitations and rules described below. Substitution is case-sensitive, and a fatal error occurs if VP is not defined when the application is saved to an IceGrid registry.

Where are Variables Allowed?

Substitution is performed in all string fields except the following:

- server and service template IDs (when defining a template or when referring to a template)
- variable names
- template parameter names
- node names
- application names

Escaping a Variable

You can prevent substitution by escaping a variable reference with an additional leading $ character. For example, in order to assign the literal string ${abc} to a variable, you would use $${abc} as this variable's value.

The extra $ symbol is only meaningful when immediately preceding a variable reference, therefore text such as US$$55 is not modified. Each occurrence of the characters $$ preceding a variable reference is replaced with a single $ character, and that character does not initiate a variable reference.

Pre-Defined Variables

IceGrid defines a set of read-only variables to hold information that may be of use to descriptors. The names of these variables are reserved and cannot be used as variable or parameter names. The table below describes the purpose of each variable and defines the context in which it is valid.

Name	Description
application	The name of the enclosing application.
application.distrib	The pathname of the enclosing application distribution directory, and an alias for ${node.datadir}/distrib/${application}.
node	The name of the enclosing node.
node.os	The name of the enclosing node operating system. On Unix, this is value is provided by uname. On Windows, the value is Windows.
node.hostname	The host name of the enclosing node.
node.release	The operation system release of the enclosing node. On Unix, this value is provided by uname. On Windows, the value is obtained from the OSVERSIONINFO data structure.
node.version	The operation system version of the enclosing node. On Unix, this value is provided by uname. On Windows, the value represents the current service pack level.
node.machine	The machine hardware name of the enclosing node. On Unix, this value is provided by uname. On Windows, the value is x86 or x64.

node.datadir	The absolute pathname of the enclosing node data directory.
server	The ID of the enclosing server.
server.distrib	The pathname of the enclosing server distribution directory, and an alias for ${node.datadir}/servers/${server}/distrib.
service	The name of the enclosing service.
session.id	The client session identifier. For sessions created with a user name and password, the value is the user ID; for sessions created from a secure connection, the value is the distinguished name associated with the connection.

The availability of a variable is easily determined in some cases, but may not be readily apparent in others. For example, you can use the ${node} variable in a property value within a server template definition, because variables in the body of a server template are evaluated when the server template is instantiated on a specific node.

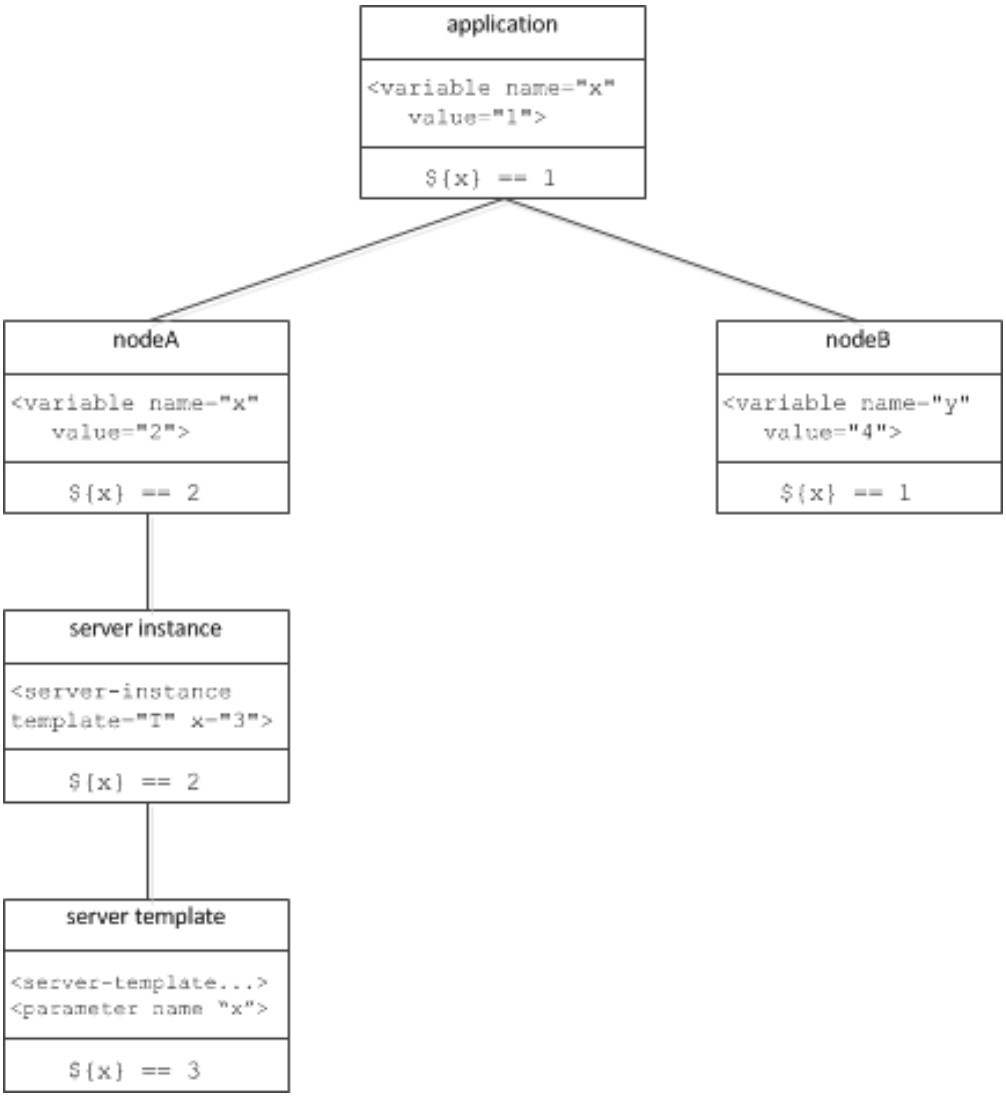
Variable Substitution in IceGrid GUI

In a number of panes, you can substitute variables and template parameters by their respective value. Use `View > Show Variables` and `View > Substitute Variables` or the corresponding toolbar toggle buttons:

When variable-substitution is enabled, the descriptors are displayed read-only.

Scoping Rules

Descriptors may only define variables at the application and node levels. Each node introduces a new scope, such that defining a variable at the node level overrides (but does not modify) the value of an application variable with the same name. Similarly, a template parameter overrides the value of a variable with the same name in an enclosing scope. A descriptor may refer to a variable defined in any enclosing scope, but its value is determined by the nearest scope. The diagram below illustrates these concepts:



In this diagram, the variable x is defined at the application level with the value 1. In node A, x is overridden with the value 2, whereas x remains unchanged in node B. Within the context of node A, x continues to have the value 2 in a server instance definition. However, when x is used as the name of a template parameter, the node's definition of x is overridden and x has the value 3 in the template's scope.

Resolving a Reference

To resolve a variable reference ${var}, IceGrid searches for a definition of var using the following order of precedence:

- Pre-defined variables
- Template parameters, if applicable
- Node variables, if applicable
- Application variables

After the initial substitution, any remaining references are resolved recursively using the following order of precedence:

- Pre-defined variables
- Node variables, if applicable
- Application variables
- Template Parameters

Template parameters are not visible in nested template instances. This situation can only occur when an IceBox server template instantiates a service template.

Modifying a Variable

A variable definition can be overridden in an inner scope, but the inner definition does not modify the outer variable.

See Also

- Using Descriptor Variables and Parameters

Application Descriptor

On this page:

Creating a new Application

You can create a new application using `File > New Application`: this opens a new Application tab, with an empty application definition.

When you are connected to an IceGrid registry, you can also use `File > New Application with Default Templates from Registry`. This also opens new Application tab with a brand new application definition. This new application contains a copy of all the templates definitions contained in the IceGrid registry default template file. See `IceGrid.Registry.DefaultTemplates`.

Properties

The Application Properties panel offers the following fields:

- **Name**
  The name of the application. This field is not editable for live applications.

- **Description**
  A free-text description of this application.

- **Variables**
  This table shows application-level IceGrid variables.

- **IcePatch2 Proxy**
  A stringified or well-known proxy for the IcePatch2 server than contains this application's distribution. Possible values:

Value	Description
"None selected" (default) or blank	no distribution defined
`${application}.IcePatch2/server`	the well-known proxy of an IcePatch2 server deployed in this application using the IcePatch2 default server-template (with a default instance-name parameter value).
stringified proxy	your own stringified or well-known proxy

- **Directories**
  List of directories included in the application distribution. When blank, the entire IcePatch2 server repository is used as the distribution.

Children

An application node can have five types of children:

- Node
- Property Set
- Replica Group
- Server Template
- Service Template

Node Descriptor

A node represents an IceGrid node that starts and monitors your servers. Several applications can be deployed on the same node; however a node descriptor describes only the servers defined by the enclosing application.

On this page:

Properties

The Node Properties panel offers the following fields:

- **Name**
  The name of the node. This name must match the IceGrid.Node.Name configuration property of the node process.

- **Description**
  A free-text description of this node.

- **Variables**
  This table shows node-level IceGrid variables.

- **Load Factor**
  A floating point value used to compare different nodes when making a load-balancing decision based on load-average (for Linux and Unix) or CPU utilization (for Windows). Leaving this value blank is equivalent to the default: 1.0 on Linux and Unix, and 1.0 divided by the number of CPUs on Windows.

Children

A node can have two types of children:

- Server
- Property Set

Server Descriptor

A server represents an Ice server deployed on a node as part of an application. IceGrid supports three kinds of servers:

- **Plain Server**
  A normal Ice server, with typically a single Ice communicator. IceGrid generates a single Ice configuration file for this server.

- **IceBox Server**
  An IceBox server, with usually a number of IceBox services deployed in it. IceGrid generates an Ice configuration file for the IceBox server itself, and also a separate Ice configuration for each IceBox service.

- **Server Instance**
  A server (either plain server or IceBox server) defined using a server template.

On this page:

Plain Server

Properties

The Server Properties panel offers the following fields:

- **Server ID**
  The ID of the server; corresponds to the `Ice.Admin.ServerId` property. Each server must have a unique ID within an IceGrid deployment.

- **Description**
  A free-text description of this server.

- **Property Sets**
  List of property-set IDs; you refer to a property set to "include" all its properties in the server.

- **Properties**
  Ice properties private to this server.

- **Log Files**
  This table can be used to declare a number of log files used by this server. Path is the path to the log file (a relative path is relative to the IceGrid node working directory); when Property is set, IceGrid generates a property with this name and the log file path as value.
  You declare log files to be able to conveniently retrieve them using IceGrid GUI (in the Live Deployment tab) or with the `icegridadmin` command-line utility.

- **Path to Executable**
  Path to the server's executable; cannot be blank. A relative path is relative to the IceGrid node working directory.

- **Ice Version**
  The Ice version of this server. If you don't provide a value, IceGrid assumes it's the same version as the IceGrid registry.

- **Working Directory**
  The working directory for the server when started by the IceGrid node.

- **Command Arguments**
  The command-line arguments given to the server when started by the IceGrid node.

- **Run as**
  On Linux and Unix, when IceGrid node is running as root, it is possible to run the server under any username. Enter the desired username in this field.

When not set (the default), the server runs as the same user as the IceGrid node, except when IceGrid node runs as root. In this case, the server runs as nobody.

- **Environment Variables**
  The environment variables for the server when started by the IceGrid node. These variables are in addition to variables defined in the IceGrid node own environment.

- **Activation Mode**
  The server's activation mode. Must be one of:

Activation Mode	Description
always	IceGrid keeps this server running all the time
manual	This server is started "manually", using IceGrid GUI or the `icegridadmin` command-line utility
on-demand	IceGrid starts this server when it resolves the object-adapter ID of an object adapter defined in this server
session	IceGrid starts a separate instance of this server for each IceGrid session that allocates this server

The Activation Mode can also be a variable or a combination of variables that resolves to one of the values above.

- **Activation Timeout**
  When activating a server, IceGrid gives timeout seconds to object adapters with server lifetime to register their endpoints with the IceGrid registry. During this time, lookup for the corresponding adapter IDs are delayed.
  If not set or set to 0, the IceGrid node uses the value of its `IceGrid.Node.WaitTime` property.

- **Deactivation Timeout**
  When deactivating a server, IceGrid gives timeout seconds to the server to exit gracefully. After this timeout, the server process is killed.
  If not set or set to 0, the IceGrid node uses the value of its `IceGrid.Node.WaitTime` property.

- **Allocatable**
  Specifies whether the server can be allocated. A server is allocated implicitly when one of its allocatable objects is allocated. This checkbox is ignored if the server activation mode is session; a server with this activation mode is always allocatable. Default: false.

- **Depends on the application distribution**
  A server that depends on the application distribution is stopped and disabled before any application distribution update, and re-enabled after such update. Default: true.

- **IcePatch2 Proxy**
  A stringified (or well-known) proxy for the IcePatch2 server than contains this server's private distribution. Possible values:

Value	Description
"None selected" (default) or blank	no private distribution defined
`${application}.IcePatch2/server`	the well-known proxy of an IcePatch2 server deployed in this application using the IcePatch2 default server-template (with a default instance-name parameter value).
stringified proxy	your own stringified or well-known proxy

- **Directories**
  List of directories included in the application distribution. When blank, the entire IcePatch2 server repository is used as the distribution.

Children

A plain server can have two types of children:

- Adapter
- Database Environment

IceBox Server

Properties

The Properties panel for an IceBox server is identical to the Properties panel for a Plain Server.

When you create a new IceBox server, some properties are created automatically:

Property Name	Property Value
IceBox.InstanceName	${server}
Ice.Admin.Endpoints	tcp -h 127.0.0.1

The `Ice.Admin.Endpoints` setting enables the Admin object in the main communicator of this IceBox server.

The Path to Executable is typically `icebox`, `iceboxd` (for a C++ IceBox), `java` (for a Java IceBox) or `iceboxcs` (for a .NET IceBox). In Java, the Command Arguments should include the IceBox container class name, `IceBox.Server`.

Children

An IceBox server can have only one type of children, Service.

Server Instance

A server instance is a server created from a server template; it may be a plain server or an IceBox server.

Properties

The Server Instance Properties panel offers the following fields:

- **Template**
  The name of the Server Template.
- **Parameters**
  Use this table to assign values to the template parameters defined in the server template.
- **Property Sets**
  List of property-set IDs; you refer to a property set to "include" all its properties in this server-instance.
- **Properties**
  Ice properties private to this server. Overall, the properties of the server instance are a combination of properties defined in the server template (including its own property sets references) augmented and possibly overridden by properties defined in the server instance.

Children

An instance of an IceBox server template can have one type of children, Property Set.

An instance of a plain server template cannot have any child.

Service Descriptor

A service represents an IceBox service deployed on an IceBox server. IceGrid supports two kinds of services:

- **Plain Service**
  A service defined directly with a service descriptor

- **Service Instance**
  A service defined using a service template

On this page:

Load Order

The load-order of services within an IceBox server is determined by their display order in IceGrid GUI. You can reorder services using `Edit > Move Up` or `Edit > Move Down` (also available from each service's contextual menu).

Plain Service

Properties

The Service Properties panel offers the following fields:

- **Service Name**
  The name of the service. Must be unique within the enclosing IceBox server.

- **Description**
  A free-text description of this service.

- **Property Sets**
  List of property-set IDs; you refer to a property set to "include" all its properties in the service.

- **Properties**
  Ice properties private to this service.

- **Log Files**
  This table can be used to declare a number of log files used by this service. Path is the path to the log file (a relative path is relative to the IceGrid node working directory); when Property is set, IceGrid generates a property with this name and the log file path as value.
  You declare log files to be able to conveniently retrieve them using IceGrid GUI (in the Live Deployment pane) or with the `icegridadmin` command-line utility.

- **Entry Point**
  The service's entry point, for example `IceStormService,35:createIceStorm`.
  IceGrid will append `--Ice.Config=`*`path-to-service-config-file`* to this entry point when it generates the `IceBox.Service.`*`name`* property in the enclosing IceBox server config file.

Children

A plain service can have two types of children:

- Adapter
- Database Environment

Service Instance

Properties

The Service Instance Properties panel offers the following fields:

- **Template**
  The name of the Service Template.

- **Parameters**
  Use this table to give values to the template parameters defined in the service template.

- **Property Sets**
  List of property-set IDs; you refer to a property set to "include" all its properties in the service.

- **Properties**
  Ice properties private to this service.

Adapter Descriptor

Each indirect object adapter registered with IceGrid requires its own Adapter descriptor. If you need to specify a direct adapter, simply create a number of Ice properties in your server.

Properties

The Adapter Properties panel offers the following fields:

- **Adapter Name**
  The name of the object adapter. This name must be unique within the enclosing Ice communicator, and is only used to lookup adapter properties.
- **Description**
  A free-text description of this object adapter.
- **Adapter ID**
  The ID of the object adapter. This ID must be unique within an IceGrid deployment. Default value: ${server}.*adapter-name*.
- **Replica Group**
  The ID of this adapter's Replica Group. By default, an adapter does not belong to any replica group.
- **Priority**
  The adapter priority in its Replica Group. The default priority is 0.
- **Endpoints**
  The configured endpoints for this object adapter. Corresponds to the *`adapter-name`*`.Endpoints` property. Default: default, which means listen using the default protocol (tcp by default) on an OS assigned port, on all interfaces.
- **Published Endpoints**
  The configured published endpoints for this object adapter. Corresponds to the *`adapter-name`*`.PublishedEndpoints` property. Default: actual endpoints (computed at run time), derived by Ice from the Endpoints field above.
- **Proxy Options**
  The default Proxy Options for well-known objects defined within this object adapter, and the Proxy Options for proxies created by this object adapter. Corresponds to the *`adapter-name`*`.ProxyEndpoints` property. Default: empty string.
- **Register Process**
  This setting is ignored for servers running Ice version 3.3 or greater.
  When checked, this adapter will create and register an `Ice::Process` object with IceGrid during activation. Such object is used to cleanly shutdown the server from IceGrid. Each server should register one, and only one such `Ice::Process` object.
- **Server Lifetime**
  When checked, IceGrid expects this adapter to register its endpoints during server startup and unregister them during server shutdown. See also Activation Timeout in Server Properties. Default: true (checked).
- **Well-known Objects**
  A table of well-known objects defined by this adapter. When Property is set, IceGrid generates a property with this name and with Identity as value.
- **Allocatable Objects**
  A table of allocatable objects defined by this adapter. When Property is set, IceGrid generates a property with this name and with Identity as value.

Database Environment Descriptor

You can describe the Freeze database environment(s) used by your server or service with Database Environment Descriptors. Each descriptor describes a single Freeze database environment.

Properties

The Database Environment Properties panel offers the following fields:

- **Name**
  The name of the Freeze Database Environment. This name must be unique within the enclosing Ice communicator.

- **Description**
  A free-text description of this database environment.

- **DB Home**
  The path of the home directory of the Berkeley DB environment. The default is "Created by the IceGrid Node": when selected, IceGrid node creates automatically a directory in its local server tree. Otherwise, if you enter your own value, the path must exist: IceGrid node will not create this directory if it does not exist.

- **Properties**
  This table shows a list of Berkeley DB properties for this database environment. These are *not* Ice properties. IceGrid generates a `DB_CONFIG` file in the DB Home directory with these settings.

Property Set Descriptor

A Property Set defines a set of Ice properties. IceGrid GUI supports two kinds of Property Sets:

- **Named Property Set**
  A property set defined within an application or within a node. Server and service definitions refer to such property sets to "include" the corresponding properties.

- **Service-Instance Property Set**
  A property set defined as a child of an IceBox server instance. Such a property set provides properties to a service instance within a concrete IceBox server.

Properties

The Property Set Properties panel offers the following fields:

- **ID** (Named Property Set only)
  The ID of a named property set.

- **Service Name** (Service Instance property set)
  The service name.

- **Property Sets**
  List of Property Set IDs. The corresponding property sets are "included" in this property set.

- **Properties**
  Ice properties private to this Property Set

Replica Group Descriptor

A replica group represents an abstract grouping of identical, or very similar, object adapters. An object adapter joins this group by setting this replica group ID in its Replica Group field.

Properties

The Replica Properties panel offers the following fields:

- **Replica Group ID**
  The ID of this replica group. Must be unique within an IceGrid deployment.
- **Description**
  A free-text description of this replica group.
- **Proxy Options**
  The default Proxy Options for well-known objects defined within this replica group.
- **Well-known Objects**
  A table of well-known objects defined by this replica group.
- **Load Balancing Policy**
  The load-balancing policy used by IceGrid when resolving the Replica Group ID in proxies. Must be one of:

Policy	Description
Adaptive	Returns the endpoints of the object adapters whose nodes report the lowest load-average (on Linux/Unix) or CPU utilization (on Windows). The actual load-average or CPU utilization value of each node is multiplied by the node's load-factor for this comparison. See Load Sample below, and Node.
Ordered	Returns the endpoints of the object adapters with the highest priority. See Adapter.
Random	Returns the endpoints of object adapters selected at random. This is the default policy.
Round-robin	IceGrid puts all the object adapters in a list, and for each new resolution, it returns the endpoints from the next *How many Adapters* items on this list.

- **How many Adapters**
  Specify the number of object adapters selected by IceGrid for each resolution. 1 is a common value. When set to 0 (the default), IceGrid returns the endpoints of all object adapters in this replica group.
- **Load Sample**
  Available only with the Adaptive load-balancing policy. Specify the load-average or CPU utilization sample to use when comparing nodes: it can be 1 (default), 5 or 15 minutes.

Server Template Descriptor

A server template is used to capture the common definitions of several similar or identical servers.

On this page:

Properties

The Server Template Properties panel offers the following fields:

- **Template ID**
  The ID the template. Must be unique within the application.

- **Parameters**
  The list of parameters for this template. Each parameter can have an optional default value.

- **Plain Server Properties**
  The remaining fields are the Plain Server fields.

Children

A plain server template can have two types of children:

- Adapter
- Database Environment

An IceBox server template can have only one kind of children, Service.

Service Template Descriptor

A service template is used to capture the common definitions of several similar or identical services.

On this page:

Properties

The Service Template Properties panel offers the following fields:

- **Template ID**
  The ID the template. Must be unique within the application.

- **Parameters**
  The list of parameters for this template. Each parameter can have an optional default value.

- **Plain Service Properties**
  The remaining fields are the Plain Service fields.

Children

A service template can have two types of children:

- Adapter
- Database Environment

IceGrid Server Activation

On this page:

Server Activation Modes

You can choose among four activation modes for servers deployed and managed by an IceGrid node:

- Manual
  You must start the server explicitly via the IceGrid GUI or `icegridadmin`, or programmatically via the `IceGrid::Admin` interface.
- Always
  IceGrid activates the server when its node starts. If the server stops, IceGrid automatically reactivates it.
- On demand
  IceGrid activates the server when a client invokes an operation on an object in the server.
- Session
  This mode also provides on-demand activation but requires the server to be allocated by a session.

Server Activation in Detail

On-demand server activation is a valuable feature of distributed computing architectures for a number of reasons:

- It minimizes application startup times by avoiding the need to pre-start all servers.
- It allows administrators to use their computing resources more efficiently because only those servers that are actually needed are running.
- It provides more reliability in the case of some server failure scenarios, e.g., the server is reactivated after a failure and may still be capable of providing some services to clients until the failure is resolved.
- It allows remote activation and deactivation.

On-demand activation occurs when an Ice client requests the endpoints of one of the server's object adapters via a locate request. If the server is not active at the time the client issues the request, the node activates the server and waits for the target object adapter to register its endpoints. Once the object adapter endpoints are registered, the registry returns the endpoint information back to the client. This sequence ensures that the client receives the endpoint information *after* the server is ready to receive requests.

Requirements for Server Activation

In order to use on-demand activation for an object adapter, the adapter must have an identifier and be entered in the IceGrid registry.

When using session activation mode, IceGrid requires that the server be allocated; on-demand activation fails for servers that have not been allocated.

The session activation mode recognizes an additional reserved variable in the server descriptor, `${session.id}`. The value of this variable is the user ID or, for SSL sessions, the distinguished name associated with the session.

Efficiency Considerations for Server Activation

Once a server is activated, it remains running indefinitely (unless it uses the session activation mode). A node deactivates a server only when explicitly requested to do so. As a result, server processes tend to accumulate on the node's host.

One of the advantages of on-demand activation is the ability to manage computing resources more efficiently. Of course there are many aspects to this, but Ice makes one technique particularly simple: servers can be configured to terminate gracefully after they have been idle for a certain amount of time.

A typical scenario involves a server that is activated on demand, used for a while by one or more clients, and then terminated automatically when no requests have been made for a configurable number of seconds. All that is necessary is setting the server's configuration property

`Ice.ServerIdleTime` to the desired idle time.

For a server activated in session activation mode, IceGrid deactivates the server when the session releases the server or when the session is destroyed.

Activating Servers with Specific User IDs

On Unix platforms you can activate server processes with specific effective user IDs, provided that the IceGrid node is running as root. If the IceGrid node does not run as root, servers are always activated with the effective user ID of the IceGrid node process. (The same is true for Windows — servers always run with the same user ID as the IceGrid node process.)

For the remainder of this section, we assume that the node runs as root on a Unix machine.

The `user` attribute of the server descriptor specifies the user ID for a server. If this attribute is not specified and the activation mode is not `session`, the default value is `nobody`. Otherwise, the default value is `${session.id}` if the activation mode is `session`.

Since individual users often have different account names and user IDs on different machines, IceGrid provides a mechanism to map the value of the `user` attribute in the server descriptor to a user account. To do this, you must configure the node to use a user account mapper object. This object must implement the `IceGrid::UserAccountMapper` interface:

Slice

```
exception UserAccountNotFoundException {}

interface UserAccountMapper
{
    string getUserAccount(string user)
            throws UserAccountNotFoundException;
}
```

The IceGrid node invokes `getUserAccount` and passes the value of the server descriptor's `user` attribute. The return value is the name of the user account.

IceGrid provides a built-in file-based user account mapper that you can configure for the node and the registry. The file contains any number of user-account-ID pairs. Each pair appears on a separate line, with white space separating the user account from the identifier. For example, the file shown below contains two entries that map two distinguished names to the user account `lisa`:

```
lisa O=ZeroC\\, Inc., OU=Ice, CN=Lisa
lisa O=ZeroC\\, Inc., OU=Ice, CN=Lisa S.
```

The distinguished names must be unique. If the same distinguished name appears several times in a file, the last entry is used.

You can specify the path of the user account file with the `IceGrid.Registry.UserAccounts` property for the registry and the `IceGrid.Node.UserAccounts` property for a node.

To configure an IceGrid node to use the IceGrid registry file-based user account mapper, you need to set the `IceGrid.Node.UserAccountMapper` property to the well-known proxy `IceGrid/RegistryUserAccountMapper`. Alternatively, you can set this property to the proxy of your own user account mapper object. Note that if this property is set, the node ignores the setting of `IceGrid.Node.UserAccounts`.

Automating Endpoint Registration

Servers must be properly configured to enable automatic endpoint registration. It should be noted however that IceGrid simplifies the configuration process in two ways:

- The IceGrid deployment facility automates the creation of a configuration file for the server, including the definition of object adapter identifiers and endpoints.
- A server that is activated automatically by an IceGrid node does not need to explicitly configure a proxy for the locator because the

IceGrid node defines it in the server's configuration file.

See Also

- Getting Started with IceGrid
- IceGrid Architecture
- Resource Allocation using IceGrid Sessions
- Server Descriptor Element
- Locator Configuration for a Server
- Using IceGrid Deployment
- IceGrid.*

IceGrid Troubleshooting

On this page:

Troubleshooting Activation Failures

Server activation failure is usually indicated by the receipt of a `NoEndpointException`. This can happen for a number of reasons, but the most likely cause is an incorrect configuration. For example, an IceGrid node may fail to activate a server because the server's executable file, shared libraries, or classes could not be found. There are several steps you can take in this case:

1. Enable activation tracing in the node by setting the configuration property `IceGrid.Node.Trace.Activator=3`.
2. Examine the tracing output and verify the server's command line and working directory are correct.
3. Relative pathnames specified in a command line may not be correct relative to the node's current working directory. Either replace relative pathnames with absolute pathnames, or restart the node in the proper working directory.
4. Verify that the server is configured with the correct `PATH` or `LD_LIBRARY_PATH` settings for its shared libraries. For a Java server, its `CLASSPATH` may also require changes.

Another cause of activation failure is a server fault during startup. After you have confirmed that the node successfully spawns the server process using the steps above, you should then check for signs of a server fault (e.g., on Unix, look for a `core` file in the node's current working directory).

Troubleshooting Proxy Failures

A client may receive `Ice::NotRegisteredException` if binding fails for an indirect proxy. This exception indicates that the proxy's object identity or object adapter is not known by the IceGrid registry. The following steps may help you discover the cause of the exception:

1. Use `icegridadmin` to verify that the object identity or object adapter identifier is actually registered, and that it matches what is used by the proxy:

```
>>> adapter list
...
>>> object find ::Hello
...
```

2. If the problem persists, review your configuration to ensure that the locator proxy used by the client matches the registry's client endpoints, and that those endpoints are accessible to the client (i.e., are not blocked by a firewall).
3. Finally, enable locator tracing in the client by setting the configuration property `Ice.Trace.Locator=2`, then run the client again to see if any log messages are emitted that may indicate the problem.

Troubleshooting Server Failures

Diagnosing a server failure can be difficult, especially when servers are activated automatically on remote hosts. Here are a few suggestions:

1. If the server is running on a Unix host, check the current working directory of the IceGrid node process for signs of a server failure, such as a `core` file.
2. Judicious use of tracing can help to narrow the search. For example, if the failure occurs as a result of an operation invocation, enable protocol tracing in the Ice run time by setting the configuration property `Ice.Trace.Protocol=1` to discover the object identity and operation name of all requests.
   Of course, the default log output channels (standard out and standard error) will probably be lost if the server is activated automatically, so either start the server manually (see below) or redirect the log output. You can also use the `Ice::Logger` interface to emit your own trace messages.
3. Run the server in a debugger; a server configured for automatic activation can also be started manually if necessary. However, since the IceGrid node did not activate the server, it cannot monitor the server process and therefore will not know when the server terminates. This will prevent subsequent activation unless you clean up the IceGrid state when you have finished debugging and

terminated the server. You can do this by starting the server using `icegridadmin`:

```
>>> server start TheServer
```

This will cause the node to activate (and therefore monitor) the server process. If you do not want to leave the server running, you can stop it with the `server stop` command.

4. After the server is activated and is in a quiescent state, attach your debugger to the running server process. This avoids the issues associated with starting the server manually (as described in the previous step), but does not provide as much flexibility in customizing the server's startup environment.

Another cause for a server to fail to activate correctly is if there is a mismatch in the adapter identifiers used by the server for its adapters, and the adapter identifiers specified in the server's deployment descriptor. After starting a server process, the node waits for the server to activate all of its object adapters and report them as ready; if the server does not do this, the node reports a failure once a timeout expires. The timeout is controlled by the setting of the property `IceGrid.Node.WaitTime`. (The default value is 60 seconds.)

You can check the status of each of a server's adapters using `icegridadmin` or the GUI tool. While the node waits for an adapter to be activated by the server, it reports the status of the adapter as "activating". If you experience timeouts before each adapter's status changes to "active", the most likely cause is that the deployment descriptor for the server either mentions more object adapters than are actually created by the server, or that the server uses an identifier for one or more adapters that does not match the corresponding identifier in the deployment descriptor.

Disabling Faulty Servers

You may find it necessary to disable a server that terminates in an error condition. For example, on a Unix platform each server failure might result in the creation of a new (and potentially quite large) core file. This problem is exacerbated when the server is used frequently, in which case repeated cycles of activation and failure can consume a great deal of disk space and threaten the viability of the application as a whole.

As a defensive measure, you can configure an IceGrid node to disable these servers automatically using the `IceGrid.Node.DisableOnFailure` property. In the disabled state, a server cannot be activated on demand. The default value of the property is zero, meaning the node does not disable a server that terminates improperly. A positive value causes the node to temporarily disable a faulty server, with the value representing the number of seconds the server should remain disabled. If the property has a negative value, the server is disabled indefinitely, or until the server is explicitly enabled or started via an administrative action.

You can also manually disable a server at any time using an administrative tool. A manually disabled server remains disabled indefinitely until an administrator enables or starts it. Disabling an *active* server has no effect on the server process; the server is unaware of the change to its status and continues to service requests from connected clients as usual. However, as of Ice 3.5, disabling a server does prevent IceGrid from including the endpoints of the server's object adapters in any subsequent locate requests, and it excludes those object adapters from any replica groups in which they might participate.

Typically, the ultimate goal of disabling a server is to gracefully migrate clients from the faulty server to ones that are behaving correctly. For a client that starts after the server is disabled, migration occurs immediately: the Ice run time in the client issues a locate request on the first proxy invocation, and the result returned by IceGrid excludes any endpoints from the disabled server. For a client that is active at the time the server is disabled, it does not migrate to a different server until its Ice run time issues a new locate request and obtains a new set of endpoints for the target object. The timing of this new locate request depends on several factors:

- If the client has an existing connection to the server, that connection will remain open and active as determined by the configuration settings of the client and server. For example, active connection management could eventually cause the connection to be closed automatically from either end due to inactivity. A subsequent proxy invocation may result in a new locate request.
- The client's use of connection caching also plays an important role. If the client's proxy is configured to cache connections, the client may continue to use the disabled server indefinitely. Connection caching should be disabled to ensure that migration eventually takes place.
- The Ice run time in the client also caches the results of locate requests. Although by default these results do not expire, setting a cache timeout allows you to specify how frequently the Ice run time issues new locate requests. Consider this code:

C++

```
proxy =
proxy->ice_connectionCached(false)->ice_locatorCacheTimeout(20);
```

By disabling connection caching and setting a locator cache timeout, we can ensure that migration occurs within twenty seconds for

invocations on this proxy.

See Also

- IceGrid Server Activation
- Locator Semantics for Clients
- icegridadmin Command Line Tool

IceGrid Database Utility

The `icegriddb` utility is a command-line tool for importing and exporting an IceGrid registry database.

On this page:

Usage

The IceGrid Database utility supports the following command-line options:

```
Usage: icegriddb <options>
Options:
 -h, --help             Show this message.
 -v, --version          Display version.
 --import FILE          Import database from FILE.
 --export FILE          Export database to FILE.
 --dbpath DIR           Source or target database environment.
 --mapsize VALUE        Set LMDB map size in MB (optional, import only).
 --server-version VER   Set Ice version for IceGrid servers (optional,
import only).
 -d, --debug            Print debug messages.
```

Exporting an IceGrid Database

To export an IceGrid registry database, use the `--export` option to specify the output file and the `--dbpath` option to specify the path name of the registry's database directory. To discover the location of your database, review the registry's configuration and look for the setting of `IceGrid.Registry.LMDB.Path`. For example, the IceGrid sample programs typically use this setting:

```
IceGrid.Registry.LMDB.Path=db/registry
```

Run the following command to export the database:

```
$ icegriddb --export registry.ixp --dbpath db/registry
```

> You can export an IceGrid registry database while the IceGrid registry is actively using this database. Write operations to the IceGrid registry database will block while `icegriddb` is reading the database.
>
> If you want to back-up the IceGrid registry database while the IceGrid registry is running, we recommend using the mdb_copy tool.

Importing an IceGrid Database

To import an IceGrid registry database, use the `--import` option to specify the input file and the `--dbpath` option to specify the path name of the registry's database directory. For example, use the following command to import a database into the `dbNew/registry` directory from a file named `registry.ixp`:

```
$ icegriddb --import registry.ixp --dbpath dbNew/registry
```

The target directory must be empty.

mapsize Option

The `--mapsize` option allows you to set the map size of the new LMDB database. See IceGrid.Registry.LMDB.MapSize for additional information.

server-version Option

IceGrid allows you to assign an Ice version to each server it manages. IceGrid uses this information to generate configuration files for this server that are compatible with the specified Ice version. When a server has no associated Ice version, IceGrid assumes this server uses the same version of Ice, for example, IceGrid 3.7.0 assumes such a server also uses Ice 3.7.0.

By default, `icegriddb` does not change this `ice-version` attribute when importing IceGrid databases: an unset attribute remains unset, and a set attribute keeps the same value. You can make `icegriddb` change all unset `ice-version` attributes to a specific version with the `--server-version` option, as shown in the example below:

```
$ icegriddb --server-version 3.5.1 --import registry.ixp --dbpath
dbNew/registry
```

Compatibility

Besides importing files that it creates itself, `icegriddb` can also import the files exported by the Ice 3.5 (`icegriddb35`) and Ice 3.6 versions of this utility.

IcePatch2

Deprecation Notice
IcePatch2 has been deprecated and will be removed in a future version of Ice.

IcePatch2 is an efficient file patching service that is easy to configure and use. It includes the following components:

- the IcePatch server (`icepatch2server`)
- a text-based IcePatch client (`icepatch2client`)
- a text-based tool to compress files and calculate checksums (`icepatch2calc`)
- a Slice API and C++ convenience library for developing custom IcePatch2 clients

As with all Ice services, IcePatch2 can be configured to use Ice facilities such as Glacier2 for firewall support and IceSSL for secure communication.

IcePatch2 is conceptually quite simple. The server is given responsibility for a file system directory (the *data directory*) containing the files and subdirectories that are to be distributed to IcePatch2 clients. You use `icepatch2calc` to compress these files and to generate an index containing a checksum for each file. The server transmits the compressed files to the client, which recreates the data directory and its contents on the client side, patching any files that have changed since the previous run.

IcePatch2 is efficient: transfer rates for files are comparable to what you would get using `ftp`.

IcePatch2 addresses a requirement common to both development and deployment scenarios: the safe, secure, and efficient replication of a directory tree. The IcePatch2 server is easy to configure and efficient. For simple uses, IcePatch2 provides a client that can be used to patch directory hierarchies from the command line. With the C++ utility library, you can also create custom patch clients if you require better integration of the client with your application.

Topics

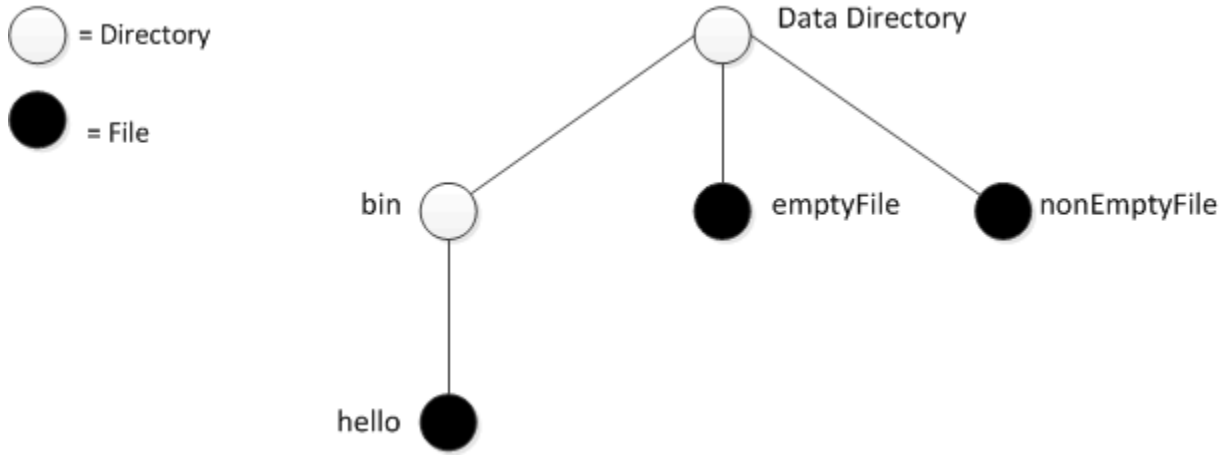
Using icepatch2calc

This page describes how to prepare a file set using `icepatch2calc`.

On this page:

Preparing a File Set using `icepatch2calc`

Suppose we have the directories and files shown below:



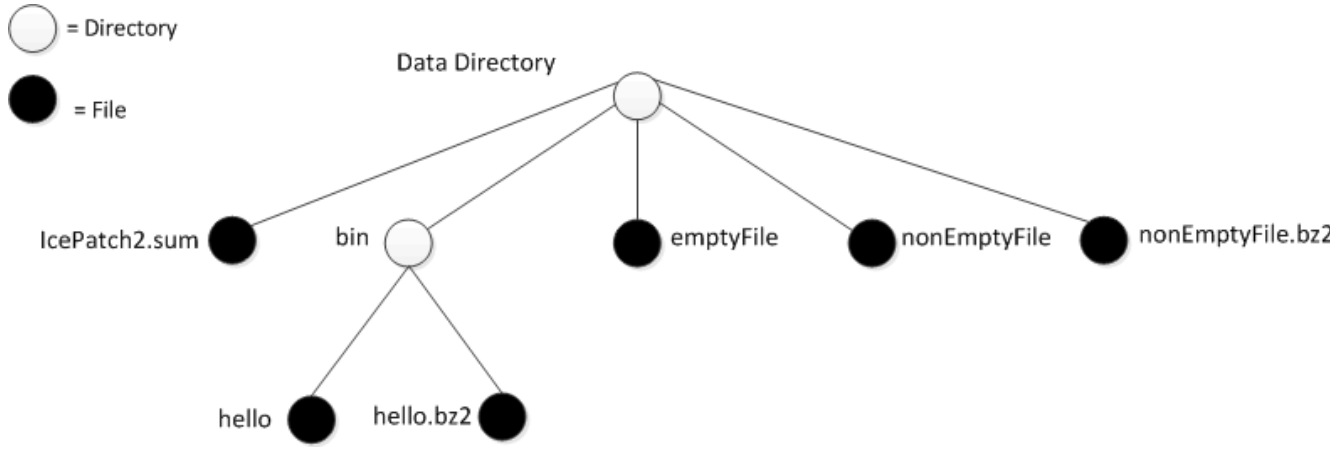
An example data directory.

Assume that the file named `emptyFile` is empty (contains zero bytes) and that the remaining files contain data.

To prepare this directory for the transmission by the server, you must first run `icepatch2calc`. (The command shown assumes that the data directory is the current directory.)

```
$ icepatch2calc .
```

After running this command, the contents of the data directory look as follows:



Contents of the data directory after running `icepatch2calc`.

Note that `icepatch2calc` compresses the files in the data directory (except for `emptyFile`, which is not compressed). Also note that `icepatch2calc` creates an additional file, `IcePatch2.sum` in the data directory. The contents of this file are as follows:

```
. 3a52ce780950d4d969792a2559cd519d7ee8c727 -1
./bin bd362140a3074eb3edb5e4657561e029092c3d91 -1
./bin/hello 77b11db586a1f20aab8553284241bb3cd532b3d5 70
./emptyFile 082c37fc2641db68d195df83844168f8a464eada 0
./nonEmptyFile aec7301c408e6ce184ae5a34e0ea46e0f0563746 72
```

Each line in the checksum file contains the name of the *uncompressed* file or directory (relative to the data directory), the checksum of the *uncompressed* file, and a byte count. For directories, the count is `-1`; for uncompressed files, the count is `0`; for compressed files, the count is the number of bytes in the *compressed* file. The lines in the file are sorted alphabetically by their pathname.

If you add files or delete files from the data directory or make changes to existing files, you must stop the server, run `icepatch2calc` again to update the `IcePatch2.sum` checksum file, and restart the server.

`icepatch2calc` Command Line Options

`icepatch2calc` has the following syntax:

```
icepatch2calc [options] data_dir [file...]
```

Normally, you will run `icepatch2calc` by simply specifying a data directory, in which case the program traverses the data directory, compresses all files, and creates an entry in the checksum file for each file and directory.

You can also nominate specific files or directories on the command line. In this case, `icepatch2calc` only compresses and calculates checksums for the specified files and directories. This is useful if you have a very large file tree and want to refresh the checksum entries for only a few selected files or directories that you have updated. (In this case, the program does not traverse the entire data directory and, therefore, will also not detect any updated, added, or deleted files, except in any of the specified directories.) Any file or directory names you specify on the command line must either be pathnames relative to the data directory or, if you use absolute pathnames, those pathnames must have the data directory as a prefix.

The command supports the following options:

- `-h, --help`
  Displays a help message.
- `-v, --version`
  Displays the version number.
- `-z, --compress`
  Normally, `icepatch2calc` scans the data directory and compresses a file only if no compressed version exists, or if the compressed version of a file has a modification time that predates that of the uncompressed version. If you specify `-z`, the tool re-scans and recompresses the entire data directory, regardless of the time stamps on files. This option is useful if you suspect that time stamps in the data directory may be incorrect.
- `-Z, --no-compress`
  This option allows you to create a client-side checksum file. Do not use this option when creating the checksum file for the server — the option is for creating a client-side `IcePatch2.sum` file for updates of software on distribution media.
- `-i, --case-insensitive`
  This option disallows file names that differ only in case. (An error message will be printed if `icepatch2calc` encounters any files that differ in case only.) This is particularly useful for Unix servers with Windows clients, since Windows folds the case of file names, and therefore such files would override each other on the Windows client.
- `-V, --verbose`
  This option prints a progress message for each file that is compressed and for each checksum that is computed.

See Also

- Running the IcePatch2 Client

Running the IcePatch2 Server

This page describes how to run the IcePatch2 server.

On this page:

Starting `icepatch2server`

Once you have run `icepatch2calc` on the data directory, you can start the `icepatch2server`:

```
$ icepatch2server .
```

The server expects the data directory as its single command-line argument. If you omit to specify the data directory, the server uses the setting of the `IcePatch2.Directory` property to determine the data directory.

You must also specify the endpoints at which the server listens for client requests, by setting the `IcePatch2.Endpoints` property.

`icepatch2server` Command Line Options

Regardless of whether you run the server under Windows or a Unix-like operating system, it provides the following options:

- `-h, --help`
  Displays a help message.
- `-v, --version`
  Displays a version number.

Additional command line options are supported, including those that allow the server to run as a Windows service or Unix daemon.

See Also

- Service Helper Class

Running the IcePatch2 Client

This page describes how to un the IcePatch2 client.

On this page:

Patching with `icepatch2client`

Once the `icepatch2server` is running, you can use `icepatch2client` to get a copy of the data directory that is maintained by the server. For example:

```
$ icepatch2client --IcePatch2Client.Proxy="IcePatch2Server/server:tcp -h
somehost.com -p 10000" .
```

The client expects the data directory as its single command-line argument. You must provide a proxy to the IcePatch2 server with the `IcePatch2Client.Proxy` property, as shown above.

If you have not run the client previously, it asks you whether you want to do a thorough patch. You must reply "yes" at this point (or run the client with the `-t` option to perform a thorough update). The client then executes the following steps:

1. It traverses the local data directory and creates a local `IcePatch2.sum` checksum file.
2. It obtains the relevant list of checksums from the server and compares it to the list of checksums it has built locally:
   - The client deletes each file that appears in the local checksum file but not in the server's file.
   - The client retrieves every file that appears in the server's checksum file, but not in the local checksum file.
   - The client patches every file that, locally, has a checksum that differs from the corresponding checksum on the server side.

When the client finishes, the contents of the data directory on the client side exactly match the contents of the data directory on the server side. However, only the uncompressed files are created on the client side — the server stores the compressed version of the files simply to avoid redundantly compressing a file every time it is retrieved by a client.

> On the initial patch, any files that exist in the client's data directory are deleted or, if they have the same name as a file on the server, will be overwritten with the corresponding file as it exists on the server.

Using `icepatch2client` for Partial Updates

Once you have run the client, the client-side data directory contains an `IcePatch2.sum` file that reflects the contents of the data directory. If you run `icepatch2client` a second time, the program uses the contents of the local checksum file: for each entry in the local checksum file, the client compares the local checksum with the server-side checksum for the same file; if the checksums differ, the client updates the corresponding file. In addition, the client deletes any files that appear in the client's checksum file but not in the server's checksum file, and it fetches any files that appear in the server's checksum file but are missing from the client's checksum file.

If you edit a client-side file and change its contents, `icepatch2client` does *not* realize that this has happened and therefore will not patch the file to be in sync with the version on the server again. This is because the client does not automatically recompute the checksum for a file to see whether the stored checksum in `IcePatch2.sum` still agrees with the actual checksum for the current file contents.

Similarly, if you create an arbitrary file on the client side, but that file is not mentioned in either the client's or the server's checksum file, that file will simply be left alone. In other words, a normal patch operates on the differences between the client's and server's checksum files, not on any differences that could be detected by examining the contents of the file system.

If you have locally created files that have nothing to do with the distribution or if you have locally modified some files and want to make sure that those modified files are updated to reflect the contents of the same files on the server side, you must run a thorough patch with the `-t` o

ption. This forces the client to traverse the local data directory and recompute the checksum for each file, and then compare these checksums against the server-side ones. As a result, if you edit a file locally so it differs from the server-side version, `-t` forces that file to be updated. Similarly, if you have added a file of your own on the client side that does not have a counterpart on the server side, that file will be deleted by a thorough patch.

Preventing Deletion of Local Files

By default, a normal patch deletes any files that appear in the client's checksum file but that are absent in the server's checksum file. Similarly, by default, a thorough patch deletes all files in the local data directory that do not appear in the server's checksum file. If you do not want this behavior, you can set the `IcePatch2Client.Remove` property to 0 (the default value is 1). This prevents deletion of files and directories that exist only on the client side, whether the patch is a normal patch or a thorough patch.

Patching Software Installed from Media

Suppose you distribute your application on a DVD that clients use to install the software. The DVD might be out of date so, after installation, the install script needs to perform a patch to update the application to the latest version. The script can perform a thorough patch to do this but, for large file sets, this is expensive because the client has to recompute the checksum for every file in the distribution.

To avoid this cost, you can place all the files for the distribution into a directory on the server and run `icepatch2calc -Z` on that directory. With the `-Z` option, `icepatch2calc` creates a checksum file with the correct checksums, but with a file size of 0 for each file, that is, the `-Z` option omits compressing the files (and the considerable cost associated with that). Once you have created the new `IcePatch2.sum` file in this way, you can include it on the DVD and install it on the client along with all the other files.

This guarantees that the checksum file on the client is in agreement with the actual files that were just installed and, therefore, it is sufficient for the install script to do a normal patch to update the distribution and so avoid the cost of recomputing the checksum for every file.

Setting Transfer Size for `icepatch2client`

You can set the `IcePatch2Client.ChunkSize` property to control the number of bytes that the client fetches per request. The default value is 100 kilobytes.

`icepatch2client` Command Line Options

The client supports the following options:

- `-h, --help`
  Displays a help message.
- `-v, --version`
  Displays a version number.
- `-t, --thorough`
  Do a thorough patch, recomputing all checksums.

See Also

- Running the IcePatch2 Server
- IcePatch2Client.*

IcePatch2 Object Identities

An IcePatch2 service hosts one well-known object, which implements the `IcePatch2::FileServer` interface and has the default identity `IcePatch2/server`. If an application requires the use of multiple IcePatch2 services, it is a good idea to assign unique identities to the well-known objects by configuring the services with different values for the `IcePatch2.InstanceName` property, as shown in the following example:

```
$ icepatch2server --IcePatch2.InstanceName=PublicFiles ...
```

This property changes the category of the server object's identity, which becomes `PublicFiles/server`.

See Also

- Object Identity
- IcePatch2.*

IcePatch2 Client Utility Library

IcePatch2 includes a pair of C++ classes that simplify the task of writing your own patch client, along with a Microsoft Foundation Classes (MFC) example that shows how to use these classes. You can find the MFC example in the subdirectory `demo/IcePatch2/MFC` of your Ice distribution.

The remainder of this section discusses the classes. To incorporate them into a custom patch client, your program must include the header file `IcePatch2/ClientUtil.h` and link with the `IcePatch2` library.

On this page:

Performing a Patch

The `Patcher` class encapsulates all of the patching logic required by a client:

C++98

```
namespace IcePatch2
{
    class Patcher : ...
    {
    public:

        Patcher(const Ice::CommunicatorPtr& communicator,
                const PatcherFeedbackPtr& feedback);

        Patcher(const FileServerPrx& server,
                const PatcherFeedbackPtr& feedback,
                const std::string& dataDir, bool thorough,
                Ice::Int chunkSize, Ice::Int remove);

        bool prepare();

        bool patch(const std::string& dir);

        void finish();
    };
    typedef IceUtil::Handle<Patcher> PatcherPtr;
}
```

Constructing a Patcher

The constructors provide two ways of configuring a `Patcher` instance. The first form obtains the following IcePatch2Client configuration properties from the supplied communicator:

- `IcePatch2Client.Proxy`
- `IcePatch2Client.Directory`
- `IcePatch2Client.Thorough`

- `IcePatch2Client.ChunkSize`
- `IcePatch2Client.Remove`

The second constructor accepts arguments that correspond to each of these properties.

Both constructors also accept a `PatcherFeedback` object, which allows the client to monitor the progress of the patch.

Executing the Patch

`Patcher` provides three methods that reflect the three stages of a patch:

- `bool prepare()`
  The first stage of a patch includes reading the contents of the checksum file (if present), retrieving the file information from the server, and examining the local data directory to compose the list of files that require updates. The `PatcherFeedback` object is notified incrementally about each local task and has the option of aborting the patch at any time. This method returns true if patch preparation completed successfully, or false if the `PatcherFeedback` object aborted the patch. If an error occurs, `prepare` raises an exception in the form of a `std::string` containing a description of the problem.

- `bool patch(const std::string& dir)`
  The second stage of a patch updates the files in the local data directory. If the `dir` argument is an empty string or `"."`, `patch` updates the entire data directory. Otherwise, `patch` updates only those files whose path names begin with the path in `dir`. For each file requiring an update, `Patcher` downloads its compressed data from the server and writes it to the local data directory. The `PatcherFeedback` object is notified about the progress of each file and, as in the preparation stage, may abort the patch if necessary. This method returns true if patching completed successfully, or false if the `PatcherFeedback` object aborted the patch. If an error occurs, `patch` raises an exception in the form of a `std::string` containing a description of the problem.

- `void finish()`
  The final stage of a patch writes a new checksum file to the local data directory. If an error occurs, `finish` raises an exception in the form of a `std::string` containing a description of the problem.

The code below demonstrates a simple patch client:

C++98

```
#include <IcePatch2/ClientUtil.h>
...
Ice::CommunicatorPtr communicator = ...;
IcePatch2::PatcherFeedbackPtr feedback = new MyPatcherFeedbackI;
IcePatch2::PatcherPtr patcher =
new IcePatch2::Patcher(communicator, feedback);

try
{
    bool aborted = !patcher->prepare();
    if(!aborted)
    {
        aborted = !patcher->patch("");
    }
    if(!aborted)
    {
        patcher->finish();
    }
    if(aborted)
    {
        cerr << "Patch aborted" << endl;
    }
}
catch(const string& reason)
{
    cerr << "Patch error: " << reason << endl;
}
```

For a more sophisticated example, see `demo/IcePatch2/MFC` in your Ice distribution.

Monitoring Patch Progress

The class `PatcherFeedback` is an abstract base class that allows you to monitor the progress of a `Patcher` object. The class declaration is shown below:

C++98

```
namespace IcePatch2
{
    class PatcherFeedback : ...
    {
    public:

        virtual bool noFileSummary(const std::string& reason) = 0;

        virtual bool checksumStart() = 0;
        virtual bool checksumProgress(const std::string& path) = 0;
        virtual bool checksumEnd() = 0;

        virtual bool fileListStart() = 0;
        virtual bool fileListProgress(Ice::Int percent) = 0;
        virtual bool fileListEnd() = 0;

        virtual bool patchStart(const std::string& path, Ice::Long size,
Ice::Long updated, Ice::Long total) = 0;
        virtual bool patchProgress(Ice::Long pos, Ice::Long size,
Ice::Long updated, Ice::Long total) = 0;
        virtual bool patchEnd() = 0;
    };
    typedef IceUtil::Handle<PatcherFeedback> PatcherFeedbackPtr;
}
```

Each of these methods returns a boolean value:

- true allows `Patcher` to continue
- false directs `Patcher` to abort the patch.

The methods are described below.

- `bool noFileSummary(const std::string& reason)`
  Invoked when the local checksum file cannot be found. Returning true initiates a thorough patch, while returning false causes `Patcher::prepare` to return false.

- `bool checksumStart()`
  `bool checksumProgress(const std::string& path)`
  `bool checksumEnd()`
  Invoked by `Patcher::prepare` during a thorough patch. The `checksumProgress` method is invoked as each file's checksum is being computed.

- `bool fileListStart()`
  `bool fileListProgress(Ice::Int percent)`
  `bool fileListEnd()`
  Invoked by `Patcher::prepare` when collecting the list of files to be updated. The `percent` argument to `fileListProgress` indicates how much of the collection process has completed so far.

- `bool patchStart(const std::string& path, Ice::Long size, Ice::Long updated, Ice::Long total)`
  `bool patchProgress(Ice::Long pos, Ice::Long size, Ice::Long updated, Ice::Long total)`
  `bool patchEnd()`
  For each file that requires updating, `Patcher::patch` invokes `patchStart` to indicate the beginning of the patch, `patchProgress` one or more times as chunks of the file are downloaded and written, and finally `patchEnd` to signal the completion of the file's patch. The `path` argument supplies the path name of the file, and `size` provides the file's compressed size. The `pos` argument

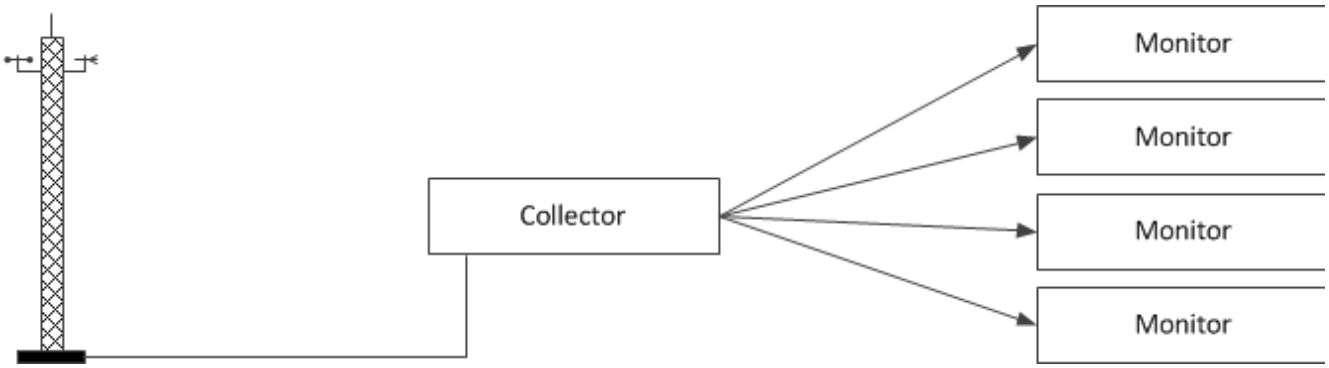
denotes the number of bytes written so far, while `updated` and `total` represent the cumulative number of bytes updated so far and the total number of bytes to be updated, respectively, of the entire patch operation.

See Also

- IcePatch2Client.*

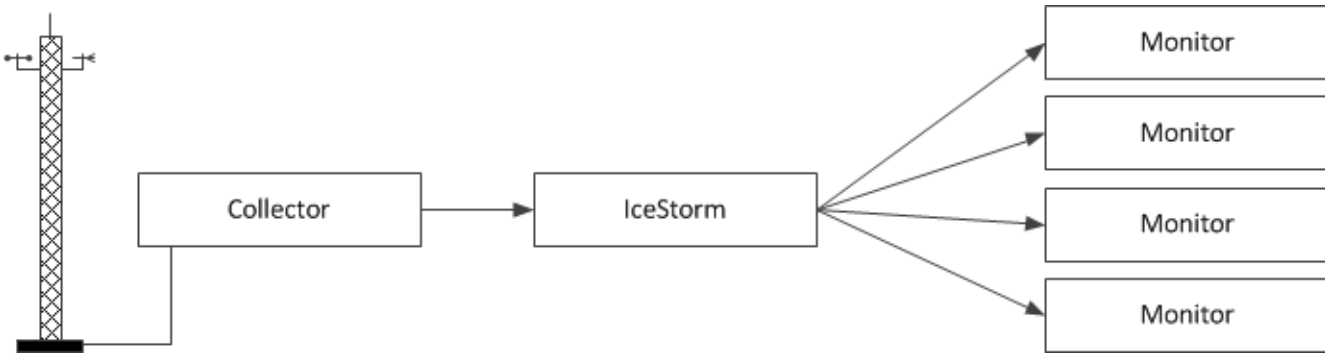
IceStorm

IceStorm is an efficient publish/subscribe service for Ice applications. Applications often need to disseminate information to multiple recipients. For example, suppose we are developing a weather monitoring application in which we collect measurements such as wind speed and temperature from a meteorological tower and periodically distribute them to weather monitoring stations. We initially consider using the architecture shown below:



Initial design for a weather monitoring application.

However, the primary disadvantage of this architecture is that it tightly couples the collector to its monitors, needlessly complicating the collector implementation by requiring it to manage the details of monitor registration, measurement delivery, and error recovery. We can rid ourselves of these mundane duties by incorporating IceStorm into our architecture, as shown below:



A weather monitoring application using IceStorm.

IceStorm simplifies the collector implementation significantly by decoupling it from the monitors. As a publish/subscribe service, IceStorm acts as a mediator between the collector (the publisher) and the monitors (the subscribers), and offers several advantages:

- When the collector is ready to distribute a new set of measurements, it makes a single request to the IceStorm server. The IceStorm server takes responsibility for delivering the request to the monitors, including handling any exceptions caused by ill-behaved or missing subscribers. The collector no longer needs to be aware of its monitors, or whether it even has any monitors at that moment.
- Similarly, monitors interact with the IceStorm server to perform tasks such as subscribing and unsubscribing, thereby allowing the collector to focus on its application-specific responsibilities and not on administrative trivia.
- The collector and monitor applications require very few changes to incorporate IceStorm.

Topics

IceStorm Concepts

This section discusses several concepts that are important for understanding IceStorm's capabilities.

- **Message**
  An IceStorm *message* is strongly typed and is represented by an invocation of a Slice operation: the operation name identifies the type of the message, and the operation parameters define the message contents. A message is published by invoking the operation on an IceStorm proxy in the normal fashion. Similarly, subscribers receive the message as a regular servant upcall. As a result, IceStorm uses the "push" model for message delivery; polling is not supported.

- **IceStorm Topics**
  An application indicates its interest in receiving messages by subscribing to a *topic*. An IceStorm server supports any number of topics, which are created dynamically and distinguished by unique names. Each topic can have multiple publishers and subscribers.

  A topic is essentially equivalent to an application-defined Slice interface: the operations of the interface define the types of messages supported by the topic. A publisher uses a proxy for the topic interface to send its messages, and a subscriber implements the topic interface (or an interface derived from the topic interface) in order to receive the messages. This is no different than if the publisher and subscriber were communicating directly in the traditional client-server style; the interface represents the contract between the client (the publisher) and the server (the subscriber), except IceStorm transparently forwards each message to multiple recipients.

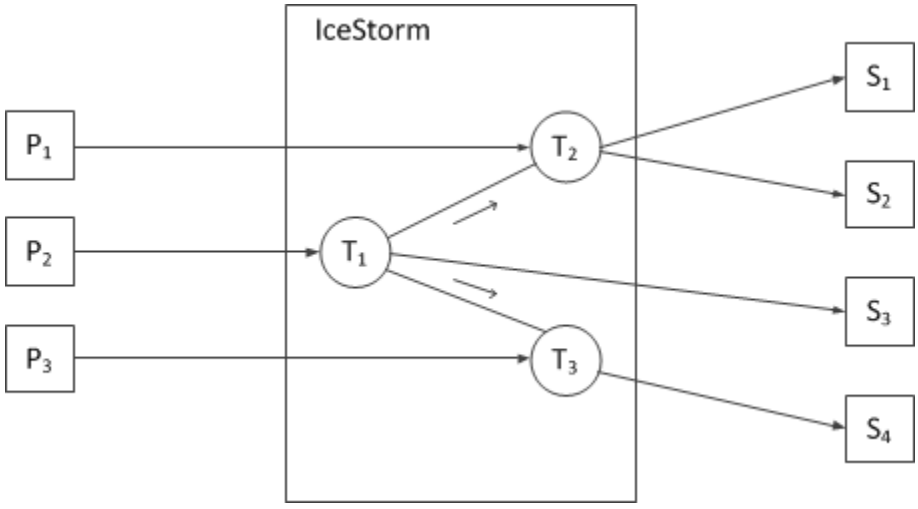
  IceStorm does not verify that publishers and subscribers are using compatible interfaces, therefore applications must ensure that topics are used correctly.

- **Unidirectional Messages**
  IceStorm messages are unidirectional, that is, they must have `void` return type, cannot have out-parameters, and cannot raise user exceptions. It follows that a publisher cannot receive replies from its subscribers. Any of the Ice transports (TCP, SSL, and UDP) can be used to publish and receive messages.

- **Federation**
  IceStorm supports the formation of topic graphs, also known as federation. A topic graph is formed by creating links between topics, where a *link* is a unidirectional association from one topic to another. Each link has a *cost* that may restrict message delivery on that link. A message published on a topic is also published on all of the topic's links for which the message cost does not exceed the link cost.

  Once a message has been published on a link, the receiving topic publishes the message to its subscribers, but does not publish it on any of its links. In other words, IceStorm messages propagate at most one hop from the originating topic in a federation.

  The following figure presents an example of topic federation. Topic T_1 has links to T_2 and T_3, as indicated by the arrows. The subscribers S_1 and S_2 receive all messages published on T_2, as well as those published on T_1. Subscriber S_3 receives messages only from T_1, and S_4 receives messages from both T_3 and T_1.



Topic federation.

  IceStorm makes no attempt to prevent a subscriber from receiving duplicate messages. For example, if a subscriber is subscribed to both T_2 and T_3, then it would receive two requests for each message published on T_1.

- **Quality of Service**
  IceStorm allows each subscriber to specify its own *quality of service* (QoS) parameters that affect the delivery of its messages. Quality of service parameters are represented as a dictionary of name-value pairs.

- **Replication**
  IceStorm supports replication to provide higher availability for publishers and subscribers.

- **Persistent Mode**
  IceStorm's default behavior maintains information about topics, links, and subscribers in a database. However, a message sent via IceStorm is not stored persistently, but rather is discarded as soon as it is delivered to the topic's current set of subscribers. If an error occurs during delivery to a subscriber, IceStorm does not queue messages for that subscriber.

- **Transient Mode**
  IceStorm can optionally run in a fully transient mode in which no database is required. Replication is not supported in this mode.

- **Subscriber Errors**

  IceStorm automatically removes a subscription from a topic if a subscriber failure occurs while attempting to deliver a message. For example, IceStorm may be unable to establish a connection to the subscriber using the proxy that the subscriber provided, meaning the subscriber is not currently active at the proxy's endpoints, or those endpoints are inaccessible to IceStorm. Another common failure scenario is a subscriber that allows an exception to propagate back to IceStorm. This is important if you make changes to a Slice data type or operation signature: if you do, you must ensure that both publishers and subscribers use the same Slice definitions; if you do not, the subscriber is likely to encounter marshaling errors when receiving an event from IceStorm with a mismatched Slice definition. If the subscriber allows this error to propagate back to IceStorm, its subscription will be canceled.

  > Use the `retryCount` quality of service parameter to configure IceStorm's behavior in error situations.

See Also

- Oneway Invocations
- Topic Federation
- IceStorm Quality of Service
- Configuring IceStorm

IceStorm Interfaces

This page provides a brief introduction to the Slice interfaces comprising the IceStorm service. See the online Slice API Reference XREF for the Slice documentation.

On this page:

The `TopicManager` Interface

The `TopicManager` is a singleton object that acts as a factory and repository of `Topic` objects. Its interface and related types are shown below:

Slice

```
module IceStorm
{
    dictionary<string, Topic*> TopicDict;

    exception TopicExists
    {
        string name;
    }

    exception NoSuchTopic
    {
        string name;
    }

    exception InvalidTopic
    {
        string reason;
    }

    interface TopicManager
    {
        Topic* create(string name) throws TopicExists, InvalidTopic;
        idempotent Topic* retrieve(string name) throws NoSuchTopic;
        idempotent TopicDict retrieveAll();
        idempotent Ice::SliceChecksumDict getSliceChecksums();
    }
}
```

The `create` operation is used to create a new topic, which must have a unique name. The `retrieve` operation allows a client to obtain a proxy for an existing topic, and `retrieveAll` supplies a dictionary of all existing topics. The `getSliceChecksums` operation returns Slice checksums for the IceStorm definitions.

The `Topic` Interface

The `Topic` interface represents a topic and provides several administrative operations for configuring links and managing subscribers.

Slice

```
module IceStorm
{
    struct LinkInfo
    {
        Topic* theTopic;
        string name;
        int cost;
    }
    sequence<LinkInfo> LinkInfoSeq;

    dictionary<string, string> QoS;

    exception LinkExists
    {
        string name;
    }

    exception NoSuchLink
    {
        string name;
    }

    exception AlreadySubscribed {}

    exception BadQoS
    {
        string reason;
    }

    interface Topic
    {
        idempotent string getName();
        idempotent Object* getPublisher();
        idempotent Object* getNonReplicatedPublisher();
        Object* subscribeAndGetPublisher(QoS theQoS, Object* subscriber)
            throws AlreadySubscribed, BadQoS;
        idempotent void unsubscribe(Object* subscriber);
        idempotent void link(Topic* linkTo, int cost)
            throws LinkExists;
        idempotent void unlink(Topic* linkTo) throws NoSuchLink;
        idempotent LinkInfoSeq getLinkInfoSeq();
        void destroy();
    }
}
```

The `getName` operation returns the name assigned to the topic, while the `getPublisher` and `getNonReplicatedPublisher` operations

return proxies for the topic's publisher object.

The `subscribeAndGetPublisher` operation adds a subscriber's proxy to the topic; if another subscriber proxy already exists with the same object identity, the operation throws `AlreadySubscribed`. The operation returns a proxy for a subscriber-specific publisher object.

The `unsubscribe` operation removes the subscriber from the topic.

A link to another topic is created using the `link` operation; if a link already exists to the given topic, the `LinkExists` exception is raised. Links are destroyed using the `unlink` operation.

Finally, the `destroy` operation permanently destroys the topic.

See Also

- Slice Checksums
- Using an IceStorm Publisher Object
- Publishing to a Specific Subscriber
- Topic Federation

Using IceStorm

Now we'll expand on the earlier weather monitoring example, demonstrating how to create, subscribe to and publish messages on a topic. We use the following Slice definitions in our example:

Slice

```
struct Measurement
{
    string tower; // tower id
    float windSpeed; // knots
    short windDirection; // degrees
    float temperature; // degrees Celsius
}

interface Monitor
{
    void report(Measurement m);
}
```

`Monitor` is our topic interface. For the sake of simplicity, it defines just one operation, `report`, taking a `Measurement` struct as its only parameter.

Topics

- Implementing an IceStorm Publisher
- Using an IceStorm Publisher Object
- Implementing an IceStorm Subscriber
- Publishing to a Specific Subscriber

See Also

- IceStorm

Implementing an IceStorm Publisher

The implementation of our weather measurement collector application can be summarized easily:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Obtain a proxy for the `Weather` topic, either by creating the topic if it does not exist, or retrieving the proxy for the existing topic.
3. Obtain a proxy for the `Weather` topic's "publisher object." This proxy is provided for the purpose of publishing messages, and therefore is narrowed to the topic interface (`Monitor`).
4. Collect and report measurements.

We present collector implementations in C++ and Java below.

On this page:

Publisher Example in C++

As usual, our C++ example begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the IceStorm Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions shown above.

C++11 C++98

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

int main(int argc, char* argv[])
{
    ...
    auto obj = communicator->stringToProxy("IceStorm/TopicManager:tcp -
p 9999");
    auto topicManager =
Ice::checkedCast<IceStorm::TopicManagerPrx>(obj);
    std::shared_ptr<IceStorm::TopicPrx> topic;
    while(!topic)
    {
        try
        {
            topic = topicManager->retrieve("Weather");
        }
        catch(const IceStorm::NoSuchTopic&)
        {
            try
            {
                topic = topicManager->create("Weather");
            }
            catch(const IceStorm::TopicExists&)
            {
                // Another client created the topic.
            }
        }
    }

    auto pub = topic->getPublisher()->ice_oneway();
    auto monitor = Ice::uncheckedCast<MonitorPrx>(pub);
    while(true)
    {
        auto m = getMeasurement();
        monitor->report(m);
    }
    ...
}
```

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy("IceStorm/TopicMan
ager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
IceStorm::TopicManagerPrx::checkedCast(obj);
    IceStorm::TopicPrx topic;
    while(!topic)
    {
        try
        {
            topic = topicManager->retrieve("Weather");
        }
        catch(const IceStorm::NoSuchTopic&)
        {
            try
            {
                topic = topicManager->create("Weather");
            }
            catch(const IceStorm::TopicExists&)
            {
                // Another client created the topic.
            }
        }
    }

    Ice::ObjectPrx pub = topic->getPublisher()->ice_oneway();
    MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
    while(true)
    {
        Measurement m = getMeasurement();
        monitor->report(m);
    }
    ...
}
```

Note that this example assumes that IceStorm uses the instance name `IceStorm`. The actual instance name may differ, and you need to use it as the category when calling `stringToProxy`.

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic:

C++11 C++98

```
std::shared_ptr<IceStorm::TopicPrx> topic;
 while(!topic)
 {
     try
     {
         topic = topicManager->retrieve("Weather");
     }
     catch(const IceStorm::NoSuchTopic&)
     {
         try
         {
             topic = topicManager->create("Weather");
         }
         catch(const IceStorm::TopicExists&)
         {
             // Another client created the topic.
         }
     }
 }
```

```
 IceStorm::TopicPrx topic;
 while(!topic)
 {
     try
     {
         topic = topicManager->retrieve("Weather");
     }
     catch(const IceStorm::NoSuchTopic&)
     {
         try
         {
             topic = topicManager->create("Weather");
         }
         catch(const IceStorm::TopicExists&)
         {
             // Another client created the topic.
         }
     }
 }
```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the `Monitor` interface. (We create a oneway proxy for the publisher purely for efficiency reasons.)

C++11 C++98

```
auto pub = topic->getPublisher()->ice_oneway();
auto monitor = Ice::uncheckedCast<MonitorPrx>(pub);
```

```
Ice::ObjectPrx pub = topic->getPublisher()->ice_oneway();
MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object:

C++11 C++98

```
while(true)
{
    auto m = getMeasurement();
    monitor->report(m);
}
```

```
while(true)
{
    Measurement m = getMeasurement();
    monitor->report(m);
}
```

Publisher Example in Java

The equivalent Java version is shown below:

Java

```
public static void main(String[] args)
{
    ...
    com.zeroc.Ice.ObjectPrx obj = communicator.stringToProxy("IceStorm/
TopicManager:tcp -p 9999");
    com.zeroc.IceStorm.TopicManagerPrx topicManager =
com.zeroc.IceStorm.TopicManagerPrx.checkedCast(obj);
    IceStorm.TopicPrx topic = null;
    while(topic == null)
    {
        try
        {
            topic = topicManager.retrieve("Weather");
        }
        catch(com.zeroc.IceStorm.NoSuchTopic ex)
        {
            try
            {
                topic = topicManager.create("Weather");
            }
            catch(com.zeroc.IceStorm.TopicExists ex)
            {
                // Another client created the topic.
            }
        }
    }

    com.zeroc.Ice.ObjectPrx pub = topic.getPublisher().ice_oneway();
    MonitorPrx monitor = MonitorPrx.uncheckedCast(pub);
    while(true)
    {
        Measurement m = getMeasurement();
        monitor.report(m);
    }
    ...
}
```

Note that this example assumes that IceStorm uses the instance name `IceStorm`. The actual instance name may differ, and you need to use it as the category when calling `stringToProxy`.

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic:

Java

```
com.zeroc.IceStorm.TopicPrx topic = null;
while(topic == null)
{
    try
    {
        topic = topicManager.retrieve("Weather");
    }
    catch(com.zeroc.IceStorm.NoSuchTopic ex)
    {
        try
        {
            topic = topicManager.create("Weather");
        }
        catch(com.zeroc.IceStorm.TopicExists ex)
        {
            // Another client created the topic.
        }
    }
}
```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the `Monitor` interface:

Java

```
com.zeroc.Ice.ObjectPrx pub = topic.getPublisher().ice_oneway();
MonitorPrx monitor = MonitorPrx.uncheckedCast(pub);
```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object:

Java

```
while(true)
{
    Measurement m = getMeasurement();
    monitor.report(m);
}
```

See Also

- Configuring IceStorm

Using an IceStorm Publisher Object

Each topic creates a publisher object for the express purpose of publishing messages. It is a special object in that it implements an Ice interface that allows the object to receive and forward requests (i.e., IceStorm messages) without requiring knowledge of the operation types.

On this page:

Type Safety Considerations for the Publisher Object

From the publisher's perspective, the publisher object appears to be an application-specific type. In reality, the publisher object can forward requests for any type, and that introduces a degree of risk: a misbehaving publisher can use `uncheckedCast` to narrow the publisher object to any type and invoke any operation; the publisher object unknowingly forwards those requests to the subscribers.

If a publisher sends a request using an incorrect type, the Ice run time in a subscriber typically responds by raising `OperationNotExistException`. However, since the subscriber receives its messages as oneway invocations, no response can be sent to the publisher object to indicate this failure, and therefore neither the publisher nor the subscriber is aware of the type-mismatch problem. In short, IceStorm places the burden on the developer to ensure that publishers and subscribers are using it correctly.

Publish using Oneway or Twoway Invocations?

IceStorm messages are unidirectional, but publishers may use either oneway or twoway invocations when sending messages to the publisher object. Each invocation style has advantages and disadvantages that you should consider when deciding which one to use. The differences between the invocation styles affect a publisher in four ways:

- Efficiency
  Oneway invocations have the advantage in efficiency because the Ice run time in the publisher does not await a reply to each message (and, of course, no reply is sent by IceStorm on the wire).
- Ordering
  The use of oneway invocations by a publisher may affect the order in which subscribers receive messages. If ordering is important, use twoway invocations with a reliability QoS of `ordered`, or use a single thread in the subscriber.
- Reliability
  Oneway invocations can be lost under certain circumstances, even when they are sent over a reliable transport such as TCP. If the loss of messages is unacceptable, or you are unable to address the potential causes of lost oneway messages, then twoway invocations are recommended.
- Delays
  A publisher may experience network-related delays when sending messages to IceStorm if subscribers are slow in processing messages. Twoway invocations are more susceptible to these delays than oneway invocations.

Selecting a Transport for the Publisher Object

Each publisher can select its own transport for message delivery, therefore the transport used by a publisher to communicate with IceStorm has no effect on how IceStorm delivers messages to its subscribers.

For example, a publisher can use a UDP transport if the possibility of lost messages is acceptable (and if IceStorm provides a UDP endpoint to publishers). However, the TCP or SSL transports are generally recommended for IceStorm's publisher endpoint in order to ensure that published messages are delivered reliably to IceStorm, even if they may not be delivered reliably to some subscribers.

Using Request Contexts with the Publisher Object

A request context is an optional argument of all remote invocations. If a publisher supplies a request context when publishing a message, IceStorm will forward it intact to subscribers.

Services such as Glacier2 employ request contexts to provide applications with more control over the service's behavior. For example, if a publisher knows that IceStorm is delivering messages to subscribers via a Glacier2 router, the publisher can influence Glacier2's behavior by including a request context, as shown in the following C++ example:

C++11 C++98

```
    auto pub = topic->getPublisher();
    Ice::Context ctx;
    ctx["_fwd"] = "Oz";
    auto monitor = Ice::uncheckedCast<MonitorPrx>(pub->ice_context(ctx)
);
```

```
    Ice::ObjectPrx pub = topic->getPublisher();
    Ice::Context ctx;
    ctx["_fwd"] = "Oz";
    MonitorPrx monitor = MonitorPrx::uncheckedCast(pub->ice_context(ctx
));
```

The `_fwd` context key, when encountered by Glacier2, causes the router to forward the request using compressed batch oneway messages. The `ice_context` proxy method is used to obtain a proxy that includes the Glacier2 request context in every invocation, eliminating the need for the publisher to specify it explicitly.

See Also

- IceStorm Quality of Service
- Oneway Invocations
- Request Contexts
- How Glacier2 uses Request Contexts
- Batched Invocations
- Proxy Methods

Implementing an IceStorm Subscriber

Our weather measurement subscriber implementation takes the following steps:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Create an object adapter to host our `Monitor` servant.
3. Instantiate the `Monitor` servant and activate it with the object adapter.
4. Subscribe to the `Weather` topic.
5. Process `report` messages until shutdown.
6. Unsubscribe from the `Weather` topic.

We present monitor implementations in C++ and Java below.

On this page:

Subscriber Example in C++

Our C++ monitor implementation begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the IceStorm Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions:

C++11 C++98

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

using namespace std;

class MonitorI : public Monitor
{
public:

    virtual void report(Measurement m, const Ice::Current&) override
    {
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << endl;
    }
};

int main(int argc, char* argv[])
{
    ...
    auto obj = communicator->stringToProxy("IceStorm/TopicManager:tcp -
p 9999");
    auto topicManager =
Ice::checkedCast<IceStorm::TopicManagerPrx>(obj);

    auto adapter = communicator->createObjectAdapter("MonitorAdapter");
```

```
auto monitor = make_shared<MonitorI>();
auto proxy = adapter->addWithUUID(monitor)->ice_oneway();
adapter->activate();

shared_ptr<IceStorm::TopicPrx> topic;
try
{
    topic = topicManager->retrieve("Weather");
    IceStorm::QoS qos;
    topic->subscribeAndGetPublisher(qos, proxy);
}
catch(const IceStorm::NoSuchTopic&)
{
    // Error! No topic found!
    ...
}

communicator->waitForShutdown();
```

```
    topic->unsubscribe(proxy);
    ...
}
```

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

using namespace std;

class MonitorI : public Monitor
{
public:

    virtual void report(const Measurement& m, const Ice::Current&)
    {
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << endl;
    }
};

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy("IceStorm/TopicMan
ager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
IceStorm::TopicManagerPrx::checkedCast(obj);

    Ice::ObjectAdapterPtr adapter = communicator->createObjectAdapter("
MonitorAdapter");

    MonitorPtr monitor = new MonitorI;
    Ice::ObjectPrx proxy = adapter->addWithUUID(monitor)->ice_oneway();
    adapter->activate();

    IceStorm::TopicPrx topic;
    try
    {
        topic = topicManager->retrieve("Weather");
        IceStorm::QoS qos;
        topic->subscribeAndGetPublisher(qos, proxy);
    }
```

```
catch(const IceStorm::NoSuchTopic&)
{
    // Error! No topic found!
    ...
}

communicator->waitForShutdown();
```

```
    topic->unsubscribe(proxy);
    ...
}
```

Our implementation of the `Monitor` servant is currently quite simple. A real implementation might update a graphical display, or incorporate the measurements into an ongoing calculation.

C++11 C++98

```
class MonitorI : public Monitor
{
public:
    virtual void report(Measurement m, const Ice::Current&) override
    {
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << endl;
    }
};
```

```
class MonitorI : public Monitor
{
public:
    virtual void report(const Measurement& m, const Ice::Current&)
    {
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << endl;
    }
};
```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the `Monitor` servant and activates it:

C++11 C++98

```
    auto adapter = communicator->createObjectAdapter("MonitorAdapter");

    auto monitor = make_shared<MonitorI>();
    auto proxy = adapter->addWithUUID(monitor)->ice_oneway();
    adapter->activate();
```

```
    Ice::ObjectAdapterPtr adapter =
communicator->createObjectAdapter("MonitorAdapter");

    MonitorPtr monitor = new MonitorI;
    Ice::ObjectPrx proxy = adapter->addWithUUID(monitor)->ice_oneway();
    adapter->activate();
```

Note that the code creates a oneway proxy for the `Monitor` servant. This is for efficiency reasons: by subscribing with a oneway proxy, IceStorm will deliver events to the subscriber via oneway messages, instead of via twoway messages. We also activate the object adapter at this time, which means the servant can now begin receiving invocations.

Next, the monitor subscribes to the topic:

C++11 C++98

```
    shared_ptr<IceStorm::TopicPrx> topic;
    try
    {
        topic = topicManager->retrieve("Weather");
        IceStorm::QoS qos;
        topic->subscribeAndGetPublisher(qos, proxy);
    }
    catch(const IceStorm::NoSuchTopic&)
    {
        // Error! No topic found!
        ...
    }
```

```
    IceStorm::TopicPrx topic;
    try
    {
        topic = topicManager->retrieve("Weather");
        IceStorm::QoS qos;
        topic->subscribeAndGetPublisher(qos, proxy);
    }
    catch(const IceStorm::NoSuchTopic&)
    {
        // Error! No topic found!
        ...
    }
```

Finally, the monitor blocks until the communicator is shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic:

C++

```
adapter->activate();
communicator->waitForShutdown();

topic->unsubscribe(proxy);
```

Subscriber Example in Java

The Java implementation of the monitor is shown below:

Java

```
class MonitorI implements Monitor
{
    @Override
    public void report(Measurement m, com.zeroc.Ice.Current curr)
    {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "   Temp: " + m.temperature + "\n");
    }
}

public static void main(String[] args)
{
    ...
    com.zeroc.Ice.ObjectPrx obj = communicator.stringToProxy("IceStorm/
TopicManager:tcp -p 9999");
    com.zeroc.IceStorm.TopicManagerPrx topicManager =
com.zeroc.IceStorm.TopicManagerPrx.checkedCast(obj);

    com.zeroc.Ice.ObjectAdapterPtr adapter =
communicator.createObjectAdapter("MonitorAdapter");

    Monitor monitor = new MonitorI();
    com.zeroc.Ice.ObjectPrx proxy =
adapter.addWithUUID(monitor).ice_oneway();
    adapter.activate();

    com.zeroc.IceStorm.TopicPrx topic = null;
    try
    {
        topic = topicManager.retrieve("Weather");
        java.util.Map<String, String> qos = null;
        topic.subscribeAndGetPublisher(qos, proxy);
    }
    catch(com.zeroc.IceStorm.NoSuchTopic ex)
    {
        // Error! No topic found!
        ...
    }

    communicator.waitForShutdown();

    topic.unsubscribe(proxy);
    ...
}
```

Our implementation of the `Monitor` servant is currently quite simple. A real implementation might update a graphical display, or incorporate

the measurements into an ongoing calculation.

Java

```
class MonitorI implements Monitor
{
    public void report(Measurement m, com.zeroc.Ice.Current curr)
    {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "   Temp: " + m.temperature + "\n");
    }
}
```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the `Monitor` servant and activates it:

Java

```
    Monitor monitor = new MonitorI();
    com.zeroc.Ice.ObjectPrx proxy =
adapter.addWithUUID(monitor).ice_oneway();
    adapter.activate();
```

Note that the code creates a oneway proxy for the `Monitor` servant. This is for efficiency reasons: by subscribing with a oneway proxy, IceStorm will deliver events to the subscriber via oneway messages, instead of via twoway messages. We also activate the object adapter at this time, which means the servant can now begin receiving invocations.

Next, the monitor subscribes to the topic:

Java

```
    com.zeroc.IceStorm.TopicPrx topic = null;
    try
    {
        topic = topicManager.retrieve("Weather");
        java.util.Map<String, String> qos = null;
        topic.subscribeAndGetPublisher(qos, proxy);
    }
    catch(com.zeroc.IceStorm.NoSuchTopic ex)
    {
        // Error! No topic found!
        ...
    }
```

Finally, the monitor blocks until the communicator is shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic:

Java

```
communicator.waitForShutdown();

topic.unsubscribe(proxy);
```

See Also

- Using IceStorm
- Oneway Invocations

Publishing to a Specific Subscriber

If you send events to the publisher object you obtain by calling `Topic::getPublisher`, the event is forwarded to all subscribers for that topic:

C++11 C++98

```
share_ptr<IceStorm::TopicPrx> topic = ...;
auto pub = topic->getPublisher()->ice_oneway();

auto monitor = Ice::uncheckedCast<MonitorPrx>(pub);
Measurement m = ...;

monitor->report(m); // Sent to all subscribers
```

```
IceStorm::TopicPrx topic = ...;
Ice::ObjectPrx pub = topic->getPublisher()->ice_oneway();

MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
Measurement m = ...;

monitor->report(m); // Sent to all subscribers
```

You can also publish an event to a single specific subscriber, by using the return value of `subscribeAndGetPublisher`. For example:

C++11 C++98

```
auto monitor = make_shared<MonitorI>();
auto proxy = adapter->addWithUUID(monitor)->ice_oneway();

std::shared_ptr<IceStorm::topicPrx> topic = ...;

Icestorm::QoS qos;
auto pub = topic->subscribeAndGetPublisher(qos, proxy);
auto monitor = Ice::uncheckedCast<MonitorPrx>(pub);

Measurement m = ...;
monitor->report(m); // Sent only to this subscriber
```

```
MonitorPtr monitor = new MonitorI;
Ice::ObjectPrx proxy = adapter->addWithUUID(monitor)->ice_oneway();

IceStorm::topicPrx topic = ...;

Icestorm::QoS qos;
Ice::ObjectPrx pub = topic->subscribeAndGetPublisher(qos, proxy);
MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);

Measurement m = ...;
monitor->report(m); // Sent only to this subscriber
```

Note that, here, we save the return value of `subscribeAndGetPublisher`. The return value is a proxy that connects specifically to the `MonitorI` instance denoted by `proxy`. However, when the code calls `report` on that proxy, instead of directly invoking on the `MonitorI` instance, the request is forwarded via IceStorm.

As it stands, this code is not very interesting. After all, the call to `monitor->report` is just a round-about way for the subscriber to publish a message to itself. However, the subscriber can pass this subscriber-specific publisher proxy to another process. When that process publishes an event via the proxy, the event is sent only to the specific subscriber, instead of to all subscribers for the topic. In turn, this is useful if you are using the observer pattern, with all observers attached to an IceStorm topic.

As an example, we might have a list whose state is to be monitored by a number of observers. Updates to the list are published to an IceStorm topic, say, `ListUpdates`. The observers of the list subscribe with an interface such as:

Slice

```
interface ListObserver
{
    void init(/* The entire state of the list */);
    void itemChange(/* The added or deleted item */);
}
```

The idea is that, when an observer first starts observing the list, the `init` operation is called on the observer and passed the entire list. This initializes the observer with the current state of the list. Thereafter, whenever the list changes, it calls `itemChange` on the observer to inform it of the addition or deletion of an item. (The details of how this happens are secondary; the important point is that the observer is informed of the current state of the list initially and, thereafter, receives incremental updates about modifications to the list, rather than the entire list whenever it changes.)

The list itself might look something like this:

Slice

```
interface List
{
    void add(Item i);
    void remove(Item i);

    void addObserver(ListObserver* lo);
    void removeObserver(ListObserver* lo);
}
```

The list provides operations to add and remove an item, as well as operations to add and remove an observer. Every time `add` or `remove` are called on the list, the list publishes an `itemChange` event to the `ListUpdates` topic; this informs all the subscribed observers of the change to the list. However, when an observer is first added, the observer's `init` operation must be called. Moreover, we want to call that method only once for each observer, so we cannot just publish the initial state of the list on a topic that all observers subscribe to.

The subscriber-specific proxy that is returned by `subscribeAndGetPublisher` solves this nicely: the implementation of `addObserver` calls `subscribeAndGetPublisher`, and then invokes `init` on the observer. This both subscribes the observer to the topic, and IceStorm forwards the call to `init` to the observer. This is preferable to the list invoking `init` on the observer directly: if the observer is misbehaved (for example, if its `init` implementation blocks for some time), the list is unaffected because IceStorm shields the list from such behavior.

See Also

- Using IceStorm

Highly Available IceStorm

IceStorm offers a highly available (HA) mode that employs master-slave replication with automatic failover in case the master fails.

On this page:

IceStorm Replication Algorithm

HA IceStorm uses the Garcia-Molina "Invitation Election Algorithm" [1] in which each replica has a priority and belongs to a replica group. The replica with the highest priority in the group becomes the coordinator, and the remaining replicas are slaves of the coordinator.

All replicas are statically configured with information about all other replicas, including their priority. The group combining works as follows:

- When recovering from an error, or during startup, replicas form a single self-coordinated group.
- Coordinators periodically attempt to combine their groups with other groups in order to form larger groups.

At regular intervals, slave replicas contact their coordinator to ensure that the coordinator is still the master of the slave's group. If a failure occurs, the replica considers itself in error and performs error recovery as described above.

Replication commences once a group contains a majority of replicas. A majority is necessary to avoid the possibility of network partitioning, in which two groups of replicas form that cannot communicate and whose database contents diverge. With respect to IceStorm, a consequence of requiring a majority is that a minimum of three replicas are necessary.

An exception to the majority rule is made during full system startup (i.e., when no replica is currently running). In this situation, replication can only commence with the participation of every replica in the group. This requirement guarantees that the databases of all replicas are synchronized, and avoids the risk that the database of an offline replica might contain more recent information.

Once a majority group has been formed, all database states are compared. The most recent database state (as determined by comparing a time stamp recorded upon each database change) is transferred to all replicas and replication commences. IceStorm is now available for use.

IceStorm Replica States

IceStorm replicas can have one of four states:

- Inactive:
  The node is inactive and awaiting an election.
- Election:
  The node is electing a coordinator.
- Reorganization:
  The replica group is reorganizing.
- Normal:
  The replica group is active and replicating.

For debugging purposes, you can obtain the state of the replicas using the `replica` command of the `icestormadmin` utility, as shown below:

```
$ icestormadmin --Ice.Config=config
>>> replica
replica count: 3
1: id:         1
1: coord:      3
1: group name: 3:191131CC-703A-41D6-8B80-D19F0D5F0410
1: state:      normal
1: group:
1: max:        3
2: id:         2
2: coord:      3
2: group name: 3:191131CC-703A-41D6-8B80-D19F0D5F0410
2: state:      normal
2: group:
2: max:        3
3: id:         3
3: coord:      3
3: group name: 3:191131CC-703A-41D6-8B80-D19F0D5F0410
3: state:      normal
3: group:      1,2
3: max:        3
```

Each line begins with the identifier of the replica. The command displays the following information:

- `id`
  The identifier of the replica.
- `coord`
  The identifier of the group's coordinator.
- `group name`
  The name of the group to which this replica belongs.
- `state`
  The replica's current state.
- `group`
  The identifiers of the other replicas in the group. Note that only the coordinator knows, or cares about, this information.
- `max`
  The maximum number of replicas seen by this replica. This value is used during startup to determine whether full participation is necessary. If the value is less than the total number of replicas, full participation is required.

Client Considerations for IceStorm Replication

As previously noted, an individual IceStorm replica can be in one of several states. However, IceStorm clients have a different perspective in which the replication group as a whole is in one of the states shown below:

- Down
  All requests to IceStorm fail.
- Inactive
  All requests to IceStorm block until the replica is either down (in which case the request fails), or becomes Active.
- Active
  Requests are processed.

It is also possible, but highly unlikely, for a request to result in an `Ice::UnknownException`. This can happen, for example, if a replica

loses the majority and thus progresses to the inactive state during request processing. In this case, the result of the request is indeterminate (the request may or may not have succeeded) and therefore the IceStorm client can draw no conclusion. The client should retry the request and be prepared for the request to fail. Consider this example:

C++11 C++98

```
shared_ptr<TopicPrx> topic = ...;
shared_ptr<Ice::ObjectPrx> sub = ...;
IceStorm::QoS qos;
topic->subscribeAndGetPublisher(qos, sub);
```

```
TopicPrx topic = ...;
Ice::ObjectPrx sub = ...;
IceStorm::QoS qos;
topic->subscribeAndGetPublisher(qos, sub);
```

The call to `subscribeAndGetPublisher` may fail in very rare cases with an `UnknownException`, indicating that the subscription may or may not have succeeded. Here is the proper way to deal with the possibility of an `UnknownException`:

C++11 C++98

```
shared_ptr<TopicPrx> topic = ...;
shared_ptr<Ice::ObjectPrx> sub = ...;
IceStorm::QoS qos;
while(true)
{
    try
    {
        topic->subscriberAndGetPublisher(qos, sub);
    }
    catch(const Ice::UnknownException&)
    {
        continue;
    }
    catch(const IceStorm::AlreadySubscribed&)
    {
        // Expected.
    }
    break;
}
```

```
TopicPrx topic = ...;
Ice::ObjectPrx sub = ...;
IceStorm::QoS qos;
while(true)
{
    try
    {
        topic->subscriberAndGetPublisher(qos, sub);
    }
    catch(const Ice::UnknownException&)
    {
        continue;
    }
    catch(const IceStorm::AlreadySubscribed&)
    {
        // Expected.
    }
    break;
}
```

Subscriber Considerations for IceStorm Replication

Subscribers can receive events from any replica. The subscriber will stop receiving events under two circumstances:

- The subscriber is unsubscribed by calling `Topic::unsubscribe`.
- The subscriber is removed as a result of a failure to deliver events. Subscribers can optionally configure a quality of service paramet er that causes IceStorm to make additional delivery attempts.

Publisher Considerations for IceStorm Replication

A publisher for HA IceStorm typically receives a proxy containing multiple endpoints. With this proxy, the publisher normally binds to a single replica and continues using that replica unless there is a failure, or until active connection management (ACM) closes the connection.

As with non-HA IceStorm, event delivery ordering can be guaranteed if the subscriber and publisher are suitably configured and the publisher continues to use the same replica when publishing events.

Ordering guarantees are lost as soon as a publisher changes to a different replica. Furthermore, a publisher may receive no notification that a change has occurred, which is possible under two circumstances:

- ACM has closed the connection.
- Publishing to a replica fails and the Ice invocation can be retried, in which case the Ice run time in the publisher automatically and transparently attempts to send the request to another replica. The publisher receives an exception if the invocation cannot be retried.

A publisher has two ways of ensuring that it is notified about a change in replicas:

- The simplest method is to use the `Topic::getNonReplicatedPublisher` operation. The proxy returned by this operation points directly at the current replica and no transparent failover to a different can occur.
- If you never want transparent failover to occur during publishing, you can configure your publisher proxy so that it contains only one endpoint. In this configuration, the `Topic::getPublisher` operation behaves exactly like `getNonReplicatedPublisher`.

Of the two strategies, using `getNonReplicatedPublisher` is preferable for two reasons:

- It does not involve changes to IceStorm's configuration.
- It is still possible to obtain a replicated publisher proxy by calling `getPublisher`, whereas if you had used the second strategy you would have eliminated that possibility.

The second strategy may be necessary in certain circumstances, such as when an existing IceStorm application is deployed and cannot be changed.

Regardless of the strategy you choose, a publisher can recover from the failure of a replica by requesting another proxy from the replicated topic using `getPublisher` or `getNonReplicatedPublisher`.

See Also

- IceStorm Administration
- IceStorm Quality of Service
- Active Connection Management
- IceStorm Delivery Modes
- Automatic Retries
- Configuring IceStorm

References

1. Garcia-Molina, H. 1982. Elections in a Distributed Computing System. *IEEE Transactions on Computers* 31 (1): 48-59.

IceStorm Administration

The IceStorm administration tool, `icestormadmin`, is a command-line program that provides administrative control of an IceStorm server.

Configuration properties inform `icestormadmin` about the topic manager(s) that you wish to administer. You have several configuration options:

- Define `IceStormAdmin.Host` and `IceStormAdmin.Port` - `icestormadmin` constructs its own proxy for the topic manager at the specified host and port
- Define `IceStormAdmin.TopicManager.Default` - specifies a proxy for the default topic manager on which administrative commands operate
- Define `IceStormAdmin.TopicManager.`*`name`* - specifies the proxies for any number of named topic managers

The tool supports the following command-line options:

```
$ icestormadmin -h
Usage: icestormadmin [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-e COMMANDS          Execute COMMANDS.
-d, --debug          Print debug messages.
```

If you specify one or more `-e` options, the tool executes the given commands and exits, otherwise the tool enters an interactive session. The `help` command displays the following usage information:

- `help`
  Print this message.
- `exit, quit`
  Exit this program.
- `create` *`TOPICS`*
  Add *`TOPICS`*.
- `destroy` *`TOPICS`*
  Remove *`TOPICS`*.
- `link` *`FROM TO`* [*`COST`*]
  Link *`FROM`* to *`TO`* with the optional *`COST`*.
- `unlink` *`FROM TO`*
  Unlink *`TO`* from *`FROM`*.
- `links` [*`INSTANCE-NAME`*]
  Without an argument, `links` displays the links of all topics in the current topic manager. You can specify a different topic manager by providing its instance name.
- `topics` [*`INSTANCE-NAME`*]
  Without an argument, *`topics`* displays the names of all topics in the current topic manager. You can specify a different topic manager by providing its instance name.
- `current` [*`INSTANCE-NAME`*]
  Set the current topic manager to the topic manager with instance name *`INSTANCE-NAME`*. The proxy of the corresponding topic manager must be specified by setting an `IceStormAdmin.TopicManager.`*`name`* property. Without an argument, the command shows the current topic manager.
- `replica` [*`INSTANCE-NAME`*]
  Display replication information for the given *`INSTANCE-NAME`*.
- `subscribers` *`TOPICS`*
  Displays the identities of the subscribers for each of the *`TOPICS`*.

Some of the commands accept one or more topic names (*TOPICS*) as arguments. Topic names containing white space or matching a command keyword must be enclosed in single or double quotes.

By default, `icestormadmin` uses the topic manager specified by your setting for `IceStormAdmin.TopicManager.Default`. For example, without additional arguments, the `create` command operates on that topic manager.

If you are using multiple topic managers, you can specify their proxies by setting `IceStormAdmin.TopicManager.`*`name`* for each topic manager. For example:

```
IceStormAdmin.TopicManager.A=A/TopicManager:tcp -h x -p 9995
IceStormAdmin.TopicManager.B=Foo/TopicManager:tcp -h x -p 9996
IceStormAdmin.TopicManager.C=Bar/TopicManager:tcp -h z -p 9995
```

This sets the proxies for three topic managers. Note that *name* need not match the instance name of the corresponding topic manager — *name* simply serves as a tag. With these property settings, the `icestormadmin` commands that accept a topic can now specify a topic manager other than the default topic manager that is configured with `IceStormAdmin.TopicManager.Default`. For example:

```
current Foo
create myTopic
create Bar/myOtherTopic
```

This sets the current topic manager to the one with instance name `Foo`; the first `create` command then creates the topic within that topic manager, whereas the second `create` command uses the topic manager with instance name `Bar`.

See Also

- Configuring IceStorm
- Highly Available IceStorm
- IceStorm Properties
- IceStormAdmin.*

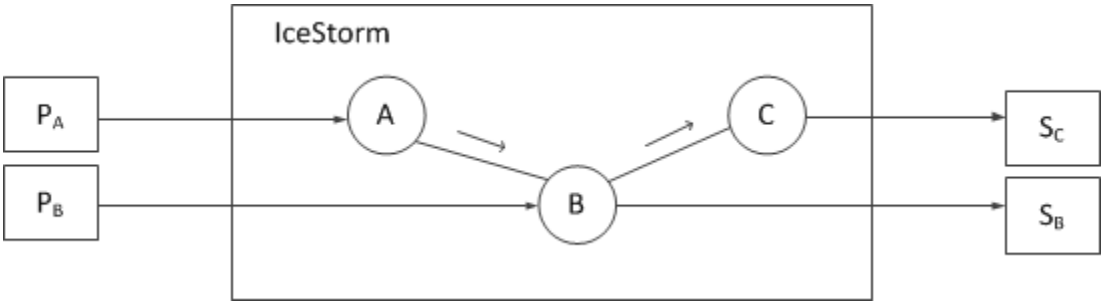
Topic Federation

The ability to link topics together into a federation provides IceStorm applications with a lot of flexibility, while the notion of a "cost" associated with links allows applications to restrict the flow of messages in creative ways. IceStorm applications have complete control of topic federation using the `TopicManager` interface described in the online XREF Slice API Reference, allowing links to be created and removed dynamically as necessary. For many applications, however, the topic graph is static and therefore can be configured using the administrative tool .

On this page:

IceStorm Message Propagation

IceStorm messages are never propagated over more than one link. For example, consider the topic graph shown below:



Message propagation.

In this case, messages published on `A` are propagated to `B`, but `B` does not propagate `A`'s messages to `C`. Therefore, subscriber S_B receives messages published on topics `A` and `B`, but subscriber S_C only receives messages published on topics `B` and `C`. If the application needs messages to propagate from `A` to `C`, then a link must be established directly between `A` and `C`.

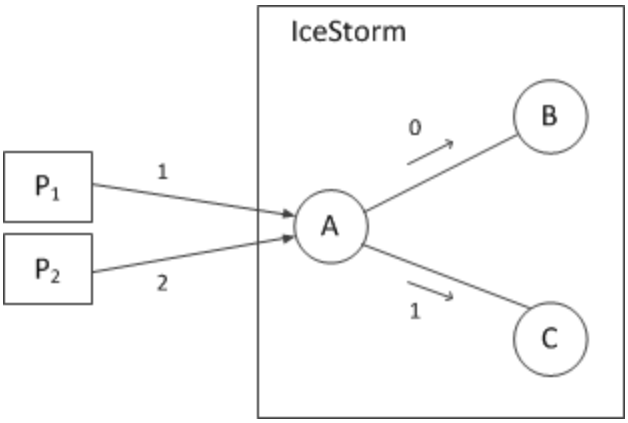
Using Cost to Limit Message Propagation

As described above, IceStorm messages are only propagated on the originating topic's immediate links. In addition, applications can use the notion of cost to further restrict message propagation.

A cost is associated with messages and links. When a message is published on a topic, the topic compares the cost associated with each of its links against the message cost, and only propagates the message on those links whose cost equals or exceeds the message cost. A cost value of zero (`0`) has the following implications:

- messages with a cost value of zero (`0`) are published on all of the topic's links regardless of the link cost;
- links with a cost value of zero (`0`) accept all messages regardless of the message cost.

For example, consider the following topic graph:



Cost semantics.

Publisher P_1 publishes a message on topic `A` with a cost of `1`. This message is propagated on the link to topic `B` because the link has a cost of `0` and therefore accepts all messages. The message is also propagated on the link to topic `C`, because the message cost does not exceed the link cost (`1`). On the other hand, the message published by P_2 with a cost of `2` is only propagated on the link to `B`.

Request Context for Cost

The cost of a message is specified in an Ice request context. Each Ice proxy operation has an implicit argument of type `Ice::Context` representing the request context. This argument is rarely used, but it is the ideal location for specifying the cost of an IceStorm message because an application only needs to supply a request context if it actually uses IceStorm's cost feature. If the request context does not contain a cost value, the message is assigned the default cost value of zero (0).

Publishing a Message with a Cost

The code examples below demonstrate how a collector can publish a measurement with a cost value of `5`. First, the C++ version:

C++

```
Measurement m = getMeasurement();
Ice::Context ctx;
ctx["cost"] = "5";
monitor->report(m, ctx);
```

And here is the equivalent version in Java:

Java

```
Measurement m = getMeasurement();
java.util.HashMap<String, String> ctx = new java.util.HashMap<>();
ctx.put("cost", "5");
monitor.report(m, ctx);
```

Receiving a Message with a Cost

A subscriber can discover the cost of a message by examining the request context supplied in the `Ice::Current` argument. For example, here is a C++ implementation of `Monitor::report` that displays the cost value if it is present:

C++11 C++98

```
    virtual void report(Measurement m, const Ice::Current& current)
override
    {
        auto p = current.ctx.find("cost");
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << "   Temp: " << m.temperature << endl;
        if(p != current.ctx.end())
        {
            cout << "   Cost: " << p->second << endl;
        }
        cout << endl;
    }
```

```
    virtual void report(const Measurement& m,
const Ice::Current& current)
    {
        Ice::Context::const_iterator p = current.ctx.find("cost");
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << "   Temp: " << m.temperature << endl;
        if(p != current.ctx.end())
        {
            cout << "   Cost: " << p->second << endl;
        }
        cout << endl;
    }
```

And here is the equivalent Java implementation:

Java

```
public void report(Measurement m, com.zeroc.Ice.Current curr)
{
    String cost = null;
    if(curr.ctx != null)
    {
        cost = curr.ctx.get("cost");
    }
    System.out.println(
        "Measurement report:\n" +
        "  Tower: " + m.tower + "\n" +
        "  W Spd: " + m.windSpeed + "\n" +
        "  W Dir: " + m.windDirection + "\n" +
        "   Temp: " + m.temperature);
    if(cost != null)
    {
        System.out.println("   Cost: " + cost);
    }
    System.out.println();
}
```

For the sake of efficiency, the Ice for Java run time may supply a null value for the request context in `Current`, therefore an application is required to check for null before using the request context.

Automating IceStorm Federation

Given the restrictions on message propagation described in the previous sections, creating a complex topic graph can be a tedious endeavor. Of course, creating a topic graph is not typically a common occurrence, since IceStorm keeps a persistent record of the graph. However, there are situations where an automated procedure for creating a topic graph can be valuable, such as during development when the graph might change significantly and often, or when graphs need to be recomputed based on changing costs.

Administration Tool Script

A simple way to automate the creation of a topic graph is to create a text file containing commands to be executed by the IceStorm administration tool. For example, the commands to create the topic graph shown earlier are shown below:

```
create A B C
link A B 0
link A C 1
```

If we store these commands in the file `graph.txt`, we can execute them using the following command:

```
$ icestormadmin --Ice.Config=config < graph.txt
```

We assume that the configuration file `config` contains the definition for the property `IceStormAdmin.TopicManager.Default`.

Proxy Considerations for IceStorm Federation

Note that, if you federate IceStorm servers, you must ensure that the proxies for the linked topics always use the same host and port (or, alternatively, can be indirectly bound via IceGrid), otherwise the federation cannot be re-established if one of the servers in the federation shuts down and is restarted later.

See Also

- IceStorm Administration
- Request Contexts
- The Current Object
- IceStorm Properties
- IceGrid

IceStorm Quality of Service

An IceStorm subscriber specifies Quality of Service (QoS) parameters at the time of subscription. The supported QoS parameters are described in the sections below

On this page:

Reliability QoS for IceStorm

The QoS parameter `reliability` affects message delivery. The only legal values at this point are `ordered` and the empty string. If not specified, the default value is the empty string (meaning not ordered).

The `ordered` reliability QoS requires a twoway subscriber proxy. If you specify this reliability QoS, IceStorm will forward events in the order they are received but doesn't forward them immediately. Instead, IceStorm waits for the reply from the forwarding of an event before forwarding the next event. This guarantees that the subscriber will process the events in the same order as they were received even if its thread pool doesn't serialize incoming requests.

Retry Count QoS for IceStorm

IceStorm automatically removes a subscriber if `ObjectNotExistException` or `NotRegisteredException` is raised while attempting to deliver an event. IceStorm considers these exceptions as indicators of a hard failure, after which it is unnecessary to continue event delivery.

For other kinds of failures, IceStorm uses the QoS parameter `retryCount` to determine when to remove a subscriber. A value of `-1` means IceStorm retries forever and never automatically removes a subscriber unless a hard failure occurs. A value of zero means IceStorm never retries and immediately removes the subscriber. For positive values, IceStorm decrements the subscriber's retry count for each failure and removes the subscriber once it reaches zero. Linked topics always have a configured retry count of `-1`. The default value of the `retryCount` parameter is zero.

A retry count of `-1` adds some resiliency to your IceStorm application by ignoring intermittent network failures such as `ConnectionRefusedException`. However, there is also some risk inherent in using a retry count of `-1` because an improperly configured subscriber may never be removed. For example, consider what happens when a subscriber registers using a transient endpoint: if that subscriber happens to terminate and resubscribe with a different endpoint, IceStorm will continue trying to deliver events to the subscriber at its old endpoint. IceStorm can only remove the subscriber if it receives a hard error, and that is only possible when the subscriber is reachable.

To use a retry count of `-1` successfully, the subscriber should either register with a fixed endpoint, or use IceGrid to take advantage of indirect proxies and automatic activation. Furthermore, if the subscriber is expected to function correctly after a restart of its process, the subscriber must use the same identity. The application can rely on the `subscribeAndGetPublisher` operation to raise `AlreadySubscribed` when the subscriber is already subscribed.

Connection caching QoS for IceStorm

The QoS parameter `connectionCached` affects the connection caching setting of the subscriber proxy used for message delivery. Defining this QoS parameter is equivalent to invoking the `ice_connectionCached` proxy method.

Locator cache timeout QoS for IceStorm

The QoS parameter `locatorCacheTimeout` affects the locator cache timeout setting of the subscriber proxy used for message delivery. Defining this QoS parameter is equivalent to invoking the `ice_locatorCacheTimeout` proxy method.

IceStorm QoS Example

The Slice type `IceStorm::QoS` is defined as a `dictionary` whose key and value types are both `string`, therefore the QoS parameter name and value are both represented as strings. The code we presented in our earlier subscriber example used an empty dictionary for the QoS argument, meaning default values are used. The C++ and Java examples shown below illustrate how to set the `reliability` parameter to `ordered`.

Here is the C++ example:

C++

```
IceStorm::QoS qos;
qos["reliability"] = "ordered";
topic->subscribeAndGetPublisher(qos, proxy->ice_twoway());
```

Here is the Java example:

Java

```
java.util.Map<String, String> qos = new java.util.HashMap<>();
qos.put("reliability", "ordered");
topic.subscribeAndGetPublisher(qos, proxy.ice_twoway());
```

See Also

- IceGrid
- Implementing an IceStorm Subscriber
- IceStorm Delivery Modes
- Object Identity

IceStorm Delivery Modes

The delivery mode for events sent to subscribers is controlled by the proxy that the subscriber passes to IceStorm. For example, if the subscriber subscribes with a oneway proxy, events will be forwarded by IceStorm as oneway messages.

On this page:

Subscribing with a Twoway Proxy

In this mode each event is sent to the subscriber as a separate twoway message. This allows the subscriber to enable server-side active connection management (ACM) without risking lost messages, because IceStorm will re-send an event if the subscriber happens to close its connection at the wrong moment.

If you combine a twoway proxy with a `reliability` QoS of `ordered`, messages will be forwarded to the subscriber in the order in which they are received. This is guaranteed because IceStorm will wait for a reply from the subscriber for each event before sending the next event.

Without ordered delivery, events may be delivered out-of-order to the subscriber because IceStorm will send an event as soon as possible (without waiting for a reply for the preceding event). If the subscriber uses a thread pool with more than one thread, this can result in out-of-order dispatch of messages in the subscriber.

For single-threaded subscribers and subscribers using a serialized thread pool, twoway delivery always produces in-order dispatch of events in the subscriber.

With twoway delivery, IceStorm is informed of any failure to deliver an event by the Ice run time. For example, IceStorm may not be able to establish a connection to a subscriber, or may receive an `ObjectNotExistException` when it forwards an event. Any failure to deliver an event to a subscriber (possibly after a transparent retry by the Ice run time) results in the cancellation of the corresponding subscription.

Subscribing with a Oneway Proxy

In this mode each event is sent to the subscriber as a oneway message. If more than one event is ready to be delivered, the events are sent in a single batch. This delivery mode is more efficient than using twoway delivery. However, the subscriber cannot use active connection management without the risk of events being lost. In addition, if something goes wrong with the subscriber, such as the subscriber having destroyed its callback object without unsubscribing, or having subscribed an object with the wrong interface, IceStorm does not notice the failure and will continue to send events to the non-existent subscriber object for as long as it can maintain a connection to the subscriber's endpoint.

For multi-threaded subscribers, oneway delivery can result in out-of-order delivery of events. For single-threaded subscribers and subscribers using a serialized thread pool, events are delivered in order.

Subscribing with a Batch Oneway Proxy

With this delivery mode, IceStorm buffers events from publishers and sends them in batches to the subscriber. This reduces network overhead and is more efficient than oneway delivery. However, as for oneway delivery, the subscriber cannot use active connection management without the risk of losing events. In addition, events can be delivered out of order if the subscriber is multi-threaded. Batch oneway delivery, while providing better throughput, increases latency because events arrive in "bursts". You can control the interval at which batched events are flushed by setting the `IceStorm.Flush.Timeout` property.

Subscribing with a Datagram Proxy

With this delivery mode, events are forwarded as UDP messages, optionally with multicast semantics. This means that events can be delivered out of order, can be lost, and can even be duplicated. In addition, IceStorm cannot detect anything about the delivery status of events. This means that if a subscriber disappears without unsubscribing, IceStorm will attempt to forward events to the subscriber

indefinitely. If you use datagram delivery, you need to be careful that subscribers unsubscribe before they disappear; otherwise, stale subscriptions can accumulate in IceStorm over time, bogging down the service as it delivers more and more events to subscribers that no longer exist.

Subscribing with a Batch Datagram Proxy

With this delivery mode, events are forwarded as batches within a datagram. The same considerations as for datagram delivery and oneway batched delivery apply here. In addition, keep in mind that, due to the size limit for datagrams, batched datagram delivery makes sense only if events are small. (You should also consider enabling compression with this delivery mode.)

See Also

- Active Connection Management
- Oneway Invocations
- The Ice Threading Model
- Batched Invocations
- IceStorm Properties

Configuring IceStorm

IceStorm is a relatively lightweight service in that it requires very little configuration and is implemented as an IceBox service. The configuration properties supported by IceStorm are described in IceStorm Properties; some of them control diagnostic output and are not discussed here.

On this page:

IceStorm Property Prefix

As you will see in IceStorm Properties, IceStorm uses its IceBox service name as the prefix for all of its properties. For example, the property *`service`*`.TopicManager.Endpoints` becomes `DemoIceStorm.TopicManager.Endpoints` when IceStorm is configured as the IceBox service `DemoIceStorm`.

IceStorm Server Configuration

The first step is configuring IceBox to run the IceStorm service:

```
IceBox.Service.DemoIceStorm=IceStormService,34:createIceStorm --Ice.Con
fig=config.service
```

In this example, the IceStorm service itself is configured by the properties in the `config.service` file, which might look as follows for a non-replicated service:

```
DemoIceStorm.LMDB.Path=db
DemoIceStorm.TopicManager.Endpoints=tcp -p 9999
DemoIceStorm.Publish.Endpoints=tcp -p 10000
```

IceStorm uses LMDB to manage the service's persistent state, therefore the first property specifies the path name of the LMDB database environment directory for the service. Here the directory `db` is used, which must already exist in the current working directory. This property can be omitted when the service is running in transient mode.

The final two properties specify the endpoints used by the IceStorm object adapters; notice that their property names begin with `DemoIceStorm`, matching the service name. The `TopicManager` property specifies the endpoints on which the `TopicManager` and `Topic` objects reside; these endpoints must use a connection-oriented protocol such as TCP or SSL. The `Publish` property specifies the endpoints used by topic publisher objects; using datagram endpoints in this property is possible but carries additional risk.

IceStorm's default thread pool configuration is sufficient when the service is running on a single CPU machine. On a host with multiple CPUs, you may be able to improve IceStorm's performance by increasing the size of its client-side thread pool using the `Ice.ThreadPool.Client.*` properties, but the optimal number of threads can only be determined with careful benchmarking.

Deploying IceStorm Replicas

There are two ways of deploying IceStorm in its highly available (replicated) mode. In both cases, adding another replica requires that all active replicas be stopped while their configurations are updated; it is not possible to add a replica while replication is running.

To remove a replica, stop all replicas and alter the configuration as necessary. You must be careful not to remove a replica if it has the latest database state. This situation will never occur during normal operation since the database state of all replicas is identical. However, in the

event of a crash it is possible for a coordinator to have later database state than all replicas. The safest approach is to verify that all replicas are active prior to stopping them. You can do this using the `icestormadmin` utility by checking that all replicas are in the `Normal` state.

IceGrid Deployment

IceGrid is a convenient way of deploying IceStorm replicas. The term *replica* is also used in the context of IceGrid, specifically when referring to groups of object adapters that participate in replication. It is important to be aware of the distinction between IceStorm replication and object adapter replication; IceStorm replication *uses* object adapter replication when deployed with IceGrid, but IceStorm does not *require* ob ject adapter replication as you will see below.

An IceGrid deployment typically uses two adapter replica groups: one for the publisher proxies, and another for the topics, as shown below:

XML

```
<replica-group id="DemoIceStorm-PublishReplicaGroup">
</replica-group>

<replica-group id="DemoIceStorm-TopicManagerReplicaGroup">
    <object identity="DemoIceStorm/TopicManager"
type="::IceStorm::TopicManager"/>
</replica-group>
```

The object adapters are then configured to use these replica groups:

XML

```
<adapter name="${service}.Publish"
    endpoints="tcp"
    replica-group="${instance-name}-PublishReplicaGroup"/>

<adapter name="${service}.TopicManager"
    endpoints="tcp"
    replica-group="${instance-name}-TopicManagerReplicaGroup"/>
```

An application may not want publisher proxies to contain multiple endpoints. In this case you should remove `PublishReplicaGroup` from the above deployment.

The next step is defining the endpoints for the adapter `Node`, which is used internally for communication with other IceStorm replicas and is not part of an adapter replica group:

XML

```
<adapter name="${service}.Node" endpoints="tcp"/>
```

Finally, you must define the node ID for each IceStorm replica using the `NodeId` property. The node ID must be a non-negative integer:

XML

```
<property name="${service}.NodeId" value="${index}"/>
```

Example
You can find a complete C++ example of an IceGrid deployment in the directory cpp/IceStorm/replicated in the ice-demos git repository.

Manual Deployment

You can also deploy IceStorm replicas without IceGrid, although it requires more manual configuration; an IceGrid deployment is simpler to maintain.

The first step is defining the set of node proxies using properties of the form `Nodes.`*`id`*. These proxies allow replicas to contact each other; their object identities are composed using *`instance-name`*`/node`*`id`*.

For example, assuming we are using the IceBox service name `IceStorm` and have three replicas with the identifiers 0, 1, 2 and an instance name of `DemoIceStorm`, we can configure the proxies as shown below:

```
IceStorm.InstanceName=DemoIceStorm
IceStorm.Nodes.0=DemoIceStorm/node0:tcp -p 13000
IceStorm.Nodes.1=DemoIceStorm/node1:tcp -p 13010
IceStorm.Nodes.2=DemoIceStorm/node2:tcp -p 13020
```

These properties must be defined in each replica. Additionally, each replica must define its node ID, as well as the node's endpoints. For example, we can configure node 0 as follows:

```
IceStorm.NodeId=0
IceStorm.Node.Endpoints=tcp -p 13000
```

The endpoints for each replica and ID must match the proxies configured in the `Nodes.`*`id`* properties.

Two additional properties allow you to configure replicated endpoints:

- *`service-name`*`.ReplicatedTopicManagerEndpoints`
  Defines the endpoints contained in proxies returned by the topic manager.
- *`service-name`*`.ReplicatedPublishEndpoints`
  Defines the endpoints contained in the publisher proxy returned by the topic.

For example, suppose we configure three replicas:

```
IceStorm.NodeId=0
IceStorm.TopicManager.Endpoints=tcp -p 10000
IceStorm.Publish.Endpoints=tcp -p 10001:udp -p 10001

IceStorm.NodeId=1
IceStorm.TopicManager.Endpoints=tcp -p 10010
IceStorm.Publish.Endpoints=tcp -p 10011:udp -p 10011

IceStorm.NodeId=2
IceStorm.TopicManager.Endpoints=tcp -p 10020
IceStorm.Publish.Endpoints=tcp -p 10021:udp -p 10021
```

Each replica should also define these properties:

```
IceStorm.ReplicatedPublishEndpoints=\

tcp -p 10001:tcp -p 10011:tcp -p 10021:udp -p 10001:udp -p 10011:udp -p
 10021
IceStorm.ReplicatedTopicManagerEndpoints=tcp -p 10000:tcp -p 10010:tcp
-p 10020
```

An application may not want publisher proxies to contain multiple endpoints. In this case you should remove the definition of the `ReplicatedPublishEndpoints` property from the above deployment.

Example
You can find a complete C++ example of a manual deployment in the directory cpp/`IceStorm/replicated2` in the `ice-demos` git repository..

IceStorm Client Configuration

Clients of the service can define a proxy for the `TopicManager` object as follows:

```
TopicManager.Proxy=IceStorm/TopicManager:tcp -p 9999
```

The name of the property is not relevant, but the endpoint must match that of the *`service`*`.TopicManager.Endpoints` property, and the object identity must use the IceStorm instance name as the category and `TopicManager` as the name.

IceStorm Object Identities

IceStorm hosts a well-known object that implements the `IceStorm::TopicManager` interface. The default identity of this object is `IceStorm/TopicManager`, as seen in the stringified proxy example above. If an application requires the use of multiple IceStorm services, it's a good idea to assign unique identities to their well-known objects by configuring the services with different values for the *`service`*`.InstanceName` property, as shown in the following example:

```
DemoIceStorm.InstanceName=Measurement
```

This property changes the category of the object's identity, which becomes `Measurement/TopicManager`. The client's configuration must also be changed to reflect the new identity:

```
TopicManager.Proxy=Measurement/TopicManager:tcp -p 9999
```

IceStorm also hosts an object with the identity `IceStorm/Finder`, as described in the next section. This identity is not affected by the service name or by changes to *`service`*`.InstanceName`.

Using the IceStorm `Finder` Interface

IceStorm supports the `IceStorm::Finder` interface:

Slice

```
module IceStorm
{
    interface Finder
    {
        TopicManager* getTopicManager();
    }
}
```

An object supporting this interface is available with the identity `IceStorm/Finder` on the service's topic manager endpoint. By knowing the host and port of this endpoint, a client can discover the topic manager's proxy at run time with a call to `getTopicManager`:

C++11 C++98

```
auto prx = communicator->stringToProxy("IceStorm/Finder:tcp -p 8888 -h
icestormhost");
auto finder = Ice::checkedCast<IceStorm::FinderPrx>(prx);
auto topicManager = finder->getTopicManager();
```

```
Ice::ObjectPrx prx = communicator->stringToProxy("IceStorm/Finder:tcp -p
8888 -h icestormhost");
IceStorm::FinderPrx finder = IceStorm::FinderPrx::checkedCast(prx);
IceStorm::TopicManagerPrx topicManager = finder->getTopicManager();
```

See Also

- IceStorm Properties
- IceBox
- IceGrid
- The Ice Threading Model
- Object Adapter Replication
- IceStorm Administration
- Using an IceStorm Publisher Object
- Highly Available IceStorm

IceStorm Persistent Data

IceStorm stores various information in a LMDB database, unless it's in transient mode. This section describes the data stored by IceStorm and database-related constraints.

On this page:

Data Stored

IceStorm stores the following data in its LMDB database:

- topics (topic name)
- subscriptions (subscriber proxy, quality of service for this subscription)
- topic links (proxy to the target topic, link cost)

Limits Imposed by the IceStorm Database

Key Size

A LMDB database consists of one or more persistent key-value maps, and the size of the keys in these maps is limited to 511 bytes.

IceStorm uses a single persistent map with application-provided data. This map has the following key:

Slice

```
struct SubscriberRecordKey
{
   // The identity of the topic, for example IceStorm/<topic-name>
   Ice::Identity topic;
   // The identity of the subscriber. An empty id corresponds to a
topic-only entry.
   Ice::Identity id;
}
```

This key is streamed into a sequence of bytes using the Ice encoding.

If you attempt to create a topic or register a subscriber with a topic and the resulting `SubscriberRecordKey`'s encoded representation is too large for the LMDB database, IceStorm will throw an `Ice::UnknownException`.

> This maximum key size is not configurable. If you exceed this limit, you need to reduce the size of your topic names or subscriber identities.

Map Size

A LMDB database has a maximum size, known as its map size. The IceStorm database can store up to `service.LMDB.MapSize` megabyt es of data in its database; any attempt to store more data will fail with an `Ice::UnknownException`. If you exceed this limit, increase `ser vice.LMDB.MapSize` and restart IceStorm.

If you don't set `service.LMDB.MapSize`, or set it to 0, IceStorm uses a map size of 10 MB on Windows, and 100 MB on Linux and

macOS.

On Windows, LMDB immediately allocates a file with the given map size, while on Linux and macOS LMDB uses sparse files and the allocated data file starts small and grows as needed, until it reaches the configured limit.

The default `MapSize` provided by IceStorm is expected to be sufficient for most applications. Unless you have a very large IceStorm deployment on Windows, we recommend keeping the default setting.

Use the `mdb_stat` utility to monitor the pages used by your IceStorm database. The example below shows a LMDB database with the default size on Linux (100 MB):

```
$ mdb_stat -e -a db
Environment Info
  Map address: (nil)
  Map size: 104857600
  Page size: 4096
  Max pages: 25600
  Number of pages used: 14
  Last transaction ID: 12
  Max readers: 126
  Number of readers used: 0
Status of Main DB
  Tree depth: 1
  Branch pages: 0
  Leaf pages: 1
  Overflow pages: 0
  Entries: 2
Status of llu
  Tree depth: 1
  Branch pages: 0
  Leaf pages: 1
  Overflow pages: 0
  Entries: 1
Status of subscribers
  Tree depth: 0
  Branch pages: 0
  Leaf pages: 0
  Overflow pages: 0
  Entries: 0
```

Backing up the IceStorm Database

You should consider making regular backups of your IceStorm database. We recommend using one of the following tools to perform backups while IceStorm is running:

- `icestormdb` with the `--export` option
- `mdb_copy` or `mdb_dump`

IceStorm Metrics

You can monitor IceStorm using the Administrative Facility and the Metrics Facet. IceStorm provides two metrics class to monitor topic and subscriber related metrics. These classes are defined in `IceStorm/Metrics.ice` and are shown below.

Slice

```
namespace IceMX
{
    class TopicMetrics extends Metrics
    {
        long published = 0;
        long forwarded = 0;
    }
    class SubscriberMetrics extends Metrics
    {
        int queued = 0;
        int outstanding = 0;
        long delivered = 0;
    }
}
```

IceStorm records metrics in the metrics map described below.

Metrics map name	Slice class	Description	Property prefix
Topic	`IceMX::TopicMetrics`	Topic metrics	`IceMX.Metrics.`*`view-name`*`.Map.Topic`
Subscriber	`IceMX::SubscriberMetrics`	Subscriber metrics	`IceMX.Metrics.`*`view-name`*`.Map.Subscriber`

To configure a metrics view to record IceStorm topic and subscriber you can use for example:

- `IceMX.Metrics.IceStormView.Map.Topic.GroupBy=id`
- `IceMX.Metrics.IceStormView.Map.Subscriber.GroupBy=id`

This will configure a view containing only the `Topic` and `Subscriber` maps. All the topics and subscribers from the IceStorm service will be monitored individually with separate metrics object.

You can use the following attributes when configuring the IceStorm `Topic` map:

Name	Description
id	The id is the topic name.
parent	The IceStorm service name.
none	The empty string.
topic	The topic name.
service	The IceStorm service name.

The `Subscriber` map can be configured with the following attributes:

Name	Description
id	The id of the subscriber metrics is the stringified proxy of the subscriber.
parent	The name of the topic name to which this subscriber belongs.

none	The empty string.
topic	The name of the topic name to which this subscriber belongs.
service	The IceStorm service name.
identity	The identity of the subscriber proxy.
facet	The facet of the subscriber proxy.
encoding	The encoding of the subscriber proxy.
mode	The mode of the subscriber proxy.
proxy	The subscriber proxy.
link	The proxy of the topic linked to the the topic which owns this subscriber.
state	The state of the subscriber. It can either be "online", "offline" or "error".

See Also

- Administrative Facility
- The Metrics Facet
- IceMX.Metrics.*

IceStorm Database Utility

The `icestormdb` utility is a command-line tool for importing and exporting IceStorm databases.

On this page:

Usage

The IceStorm Database utility supports the following command-line options:

```
Usage: icestormdb <options>
Options:
 -h, --help            Show this message.
 -v, --version         Display version.
 --import FILE         Import database from FILE.
 --export FILE         Export database to FILE.
 --dbpath DIR          Source or target database environment.
 --mapsize VALUE       Set LMDB map size in MB (optional, import only).
 -d, --debug           Print debug messages.
```

Exporting an IceStorm Database

To export an IceStorm database, use the `--export` option to specify the output file and the `--dbpath` option to specify the path name of the database. For example, use the following command to export a database found in the `db` directory to a file named `db.ixp`:

```
$ icestormdb --export db.ixp --dbpath db
```

> You can export an IceStorm database while IceStorm is actively using this database. Write operations to the IceStorm database will block while icestormdb is reading the database.
>
> If you want to back-up the IceStorm database while IceStorm is running, we recommend using the mdb_copy tool.

Importing an IceStorm Database

To import an IceStorm database, use the `--import` option to specify the input file and the `--dbpath` option to specify the path name of the database. For example, use the following command to import a database into the `dbNew` directory from a file named `db.ixp`:

```
$ icestormdb --import db.ixp --dbpath dbNew
```

The target directory must be empty.

mapsize Option

The `--mapsize` option allows you to set the map size of the new LMDB database. See `service.LMDB.MapSize` for additional information.

Compatibility

Besides importing files that it creates itself, `icestormdb` can also import the files exported by the Ice 3.5 (`icestormdb35`) and Ice 3.6 versions of this utility.

The Ice Protocol

The Ice protocol definition consists of three major parts:

- a set of data encoding rules that determine how the various data types are serialized
- a number of message types that are interchanged between client and server, together with rules as to what message is to be sent under what circumstances
- a set of rules that determine how client and server agree on a particular protocol and encoding version

Topics

Data Encoding

The key goals of the Ice data encoding are simplicity and efficiency. In keeping with these principles, the encoding does not align primitive types on word boundaries and therefore eliminates the wasted space and additional complexity that alignment requires. The Ice data encoding simply produces a stream of contiguous bytes; data contains no padding bytes and need not be aligned on word boundaries.

Data is always encoded using little-endian byte order for numeric types. (Most machines use a little-endian byte order, so the Ice data encoding is "right" more often than not.) Ice does not use a "receiver makes it right" scheme because of the additional complexity this would introduce. Consider, for example, a chain of receivers that merely forward data along the chain until that data arrives at an ultimate receiver. (Such topologies are common for event distribution services.) The Ice protocol permits all the intermediates to forward the data without requiring it to be unmarshaled: the intermediates can forward requests by simply copying blocks of binary data. With a "receiver makes it right" scheme, the intermediates would have to unmarshal and remarshal the data whenever the byte order of the next receiver in the chain differs from the byte order of the sender, which is inefficient.

Ice requires clients and servers that run on big-endian machines to incur the extra cost of byte swapping data into little-endian layout, but that cost is insignificant compared to the overall cost of sending or receiving a request.

Topics

See Also

- Protocol Messages

Basic Data Encoding

On this page:

Encoding for Sizes

Many of the types involved in the data encoding, as well as several protocol message components, have an associated size or count. A size is a non-negative number. Sizes and counts are encoded in one of two ways:

1. If the number of elements is less than 255, the size is encoded as a single `byte` indicating the number of elements.
2. If the number of elements is greater than or equal to 255, the size is encoded as a `byte` with value `255`, followed by an `int` indicating the number of elements.

Using this encoding to indicate sizes is significantly cheaper than always using an `int` to store the size, especially when marshaling sequences of short strings: counts of up to 254 require only a single byte instead of four. This comes at the expense of counts greater than 254, which require five bytes instead of four. However, for sequences or strings of length greater than 254, the extra byte is insignificant.

Encoding for Encapsulations

An encapsulation is used to contain variable-length data that an intermediate receiver may not be able to decode, but that the receiver can forward to another recipient for eventual decoding. An encapsulation is encoded as if it were the following structure:

Slice

```
struct Encapsulation
{
    int size;
    byte major;
    byte minor;
    // [... size - 6 bytes ...]
}
```

The `size` member specifies the size of the encapsulation in bytes (including the `size`, `major`, and `minor` fields). The `major` and `minor` fields specify the encoding version of the data contained in the encapsulation. The version information is followed by `size-6` bytes of encoded data.

All the data in an encapsulation is context-free, that is, nothing inside an encapsulation can refer to anything outside the encapsulation. This property allows encapsulations to be forwarded among address spaces as a blob of data.

Encapsulations can be nested, that is, contain other encapsulations.

An encapsulation can be empty, in which case the value of `size` is 6.

Encoding for Slices

The encoding format of slices changed in version 1.1.

Slice Encoding version 1.0

Exceptions and classes may be subject to *slicing* if the receiver of a value only partially understands the received value (that is, only has knowledge of a base type, but not of the actual run-time derived type). To allow the receiver of an exception or class to ignore those parts of a value that it does not understand, exception and class values are marshaled as a sequence of slices (one slice for each level of the inheritance hierarchy). A slice is a byte count encoded as a fixed-length four-byte integer, followed by the data for the slice. (The byte count includes the four bytes occupied by the count itself, so an empty slice has a byte count of four and no data.) The receiver of a value can skip over a slice by reading the byte count *b*, and then discarding the next *b-4* bytes in the input stream.

Slice Encoding version 1.1

Version 1.1 of the encoding still marshals exceptions and classes as slices in conceptually the same manner as for version 1.0, but bit flags in the leading byte of each slice determine its format and content.

Type ID

The initial slice of a class or exception, representing the instance's most-derived type, always includes a type ID. For an exception, the type ID in the initial slice is encoded as a string. For a class, the type ID in the initial slice can either be encoded as a string, an index (if the same type ID has already been encoded in the current encapsulation), or a compact ID.

Whether any subsequent slices include some form of type ID depends on the format with which the value was encoded: to facilitate slicing an instance to a less-derived type, the sliced format includes a type ID in every slice, whereas the compact format excludes type IDs in subsequent slices to conserve space while sacrificing the slicing feature.

Optional Data Members

This flag is true if the slice includes any optional data members, which are encoded after all required members. If a slice encodes its size, the size includes the optional data members.

Object Indirection Table

This flag can only be true when using the sliced format. In this case, data members that refer to class instances are encoded as indices into an indirection table that immediately follows the slice. The slice's size does *not* include the indirection table. This flag should only be set to true when there is at least one non-nil object reference in the slice, that is, when the indirection table is not empty.

Slice Size

If this flag is true, a byte count encoded as a fixed-length four-byte integer immediately follows the type ID. (The byte count includes the four bytes occupied by the count itself, so an empty slice has a byte count of four and no data.) The byte count indicates the number of bytes occupied by the encoding for the required and optional data members, but does not include the size of an object indirection table, if present.

A receiver can skip over a slice by reading the byte count *b*, and then discarding the next *b-4* bytes in the input stream. If an object indirection table is present, the receiver must then decode the table.

If this flag is false, it implies that the sender used the compact format and therefore skipping slices is not possible. The receiver must know the most-derived type in this situation otherwise decoding will fail.

Last Slice

This flag indicates whether the current slice is the last slice of the instance.

Slice Flags

The table below shows how to interpret the bit flags in the leading byte of a slice:

Bit number	Description
0-1	0 = no type ID is encoded for the slice
	1 = type ID is encoded as a string
	2 = type ID is an index encoded as a size
	3 = type ID is a compact ID encoded as a size
2	Whether or not the slice contains optional data members
3	Whether or not the slice contains an indirection table
4	Whether or not the slice size follows the type ID
5	If 0, more slices will follow, if 1, this is the last slice
6	Reserved for future use

7	Reserved for future use

Bit flags for a slice.

Encoding for Basic Types

The basic types are encoded as shown in the table. Integer types (`short`, `int`, `long`) are represented as two's complement numbers, and floating point types (`float`, `double`) use the IEEE standard formats [1]. All numeric types use a little-endian byte order.

Type	Encoding
`bool`	A single byte with value `1` for `true`, `0` for `false`
`byte`	An uninterpreted byte
`short`	Two bytes (LSB, MSB)
`int`	Four bytes (LSB .. MSB)
`long`	Eight bytes (LSB .. MSB)
`float`	Four bytes (23-bit fractional mantissa, 8-bit exponent, sign bit)
`double`	Eight bytes (52-bit fractional mantissa, 11-bit exponent, sign bit)

Encoding for basic types.

Encoding for Strings

Strings are encoded as a size, followed by the string contents in UTF-8 format [2]. Strings are not NUL-terminated. An empty string is encoded with a size of zero.

Encoding for Sequences

Sequences are encoded as a size representing the number of elements in the sequence, followed by the elements encoded as specified for their type.

Encoding for Dictionaries

Dictionaries are encoded as a size representing the number of key-value pairs in the dictionary, followed by the pairs. Each key-value pair is encoded as if it were a `struct` containing the key and value as members, in that order.

Encoding for Enumerators

The encoding format of enumerators changed in version 1.1.

Enumerator Encoding version 1.0

The number of bytes required to encode an enumerator in version 1.0 is determined by the largest value in the enumeration. In enumerations with no custom enumerator values, the largest value is the number of enumerators less one; the value encoded for an enumerator is its ordinal value in the definition, with the first enumerator having the value zero. For example, the largest value in the following enumeration is 2:

Slice

```
enum Fruit { Apple, Pear, Orange } // Encoded values: Apple = 0, Pear =
1, Orange = 2
```

When an enumeration includes custom values, the encoding uses the enumerator's assigned value, which is not necessarily the same as its ordinal value. Consider this example:

Slice

```
enum Fruit { Apple = 1, Pear = 3, Orange } // Encoded values: Apple = 1,
Pear = 3, Orange = 4
```

The largest value in this case is 4.

An enumerator is encoded as follows:

- If the largest value is in the range 0..126, the enumerator's value is marshaled as a `byte`.
- If the largest value is in the range 127..32766, the enumerator's value is marshaled as a `short`.
- If the largest value is greater than 32766, the enumerator's value is marshaled as an `int`.

> Changing the definition of an enumeration can break compatibility with existing applications. For example, if enumerators are added, removed, or changed such that the largest value crosses one of the thresholds shown above, the encoded form of the enumerators will change and cause marshaling errors unless you rebuild all applications that use this definition.

Enumerator Encoding version 1.1

An enumerator is encoded as a size, meaning the encoding of an enumerator requires one byte if its value is less than 255, or five bytes if its value is 255 or greater. The encoding uses the Slice value of the enumerator, which is not necessarily the same as its ordinal value. Repeating the previous example:

Slice

```
enum Fruit { Apple = 1, Pear = 3, Orange } // Encoded values: Apple = 1,
Pear = 3, Orange = 4
```

Although enumerator `Pear` may have ordinal value 1 in some language mappings (notably, Java), the encoding uses its Slice value of 3.

Encoding for Structures

The members of a structure are encoded in the order they appear in the `struct` declaration, as specified for their types.

See Also

- Protocol Messages
- Protocol and Encoding Versions
- Data Encoding for Exceptions
- Data Encoding for Classes

References

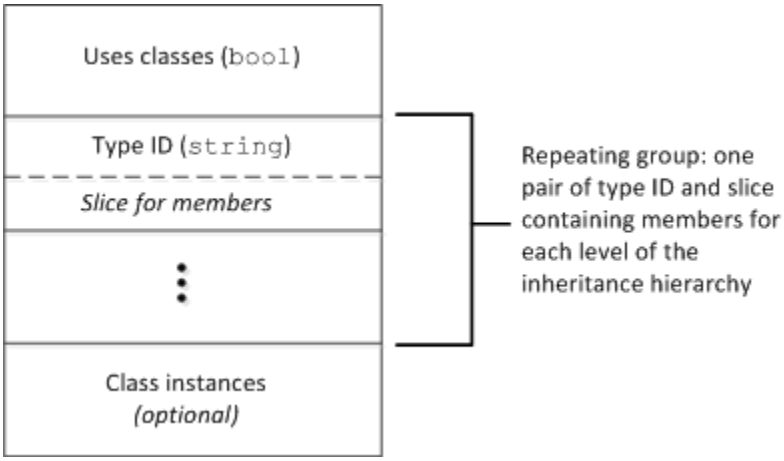
1. Institute of Electrical and Electronics Engineers. 1985. *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*. Piscataway, NJ: Institute of Electrical and Electronic Engineers.

2. Unicode Consortium, ed. 2000. *The Unicode Standard, Version 3.0*. Reading, MA: Addison-Wesley.

Data Encoding for Exceptions

Exception Encoding version 1.0

An exception is marshaled as shown below:



Marshaling format for exceptions.

Every exception instance is preceded by a single byte that indicates whether the exception uses class members: the byte value is `1` if any of the exception members are classes (or if any of the exception members, recursively, contain class members) and `0`, otherwise.

Following the header byte, the exception is marshaled as a sequence of pairs: the first member of each pair is the type ID for an exception slice, and the second member of the pair is a slice containing the marshaled members of that slice. The sequence of pairs is marshaled in derived-to-base order, with the most-derived slice first, and ending with the least-derived slice. Within each slice, data members are marshaled as for structures: in the order in which they are defined in the Slice definition.

Following the sequence of pairs, any class instances that are used by the members of the exception are marshaled. This final part is optional: it is present only if the header byte is `1`.

To illustrate the marshaling, consider the following exception hierarchy:

Slice

```
exception Base
{
    int baseInt;
    string baseString;
}

exception Derived extends Base
{
    bool derivedBool;
    string derivedString;
    double derivedDouble;
}
```

Assume that the exception members are initialized to the values shown below:

Member	Type	Value	Marshaled size (in bytes)
`baseInt`	`int`	`99`	`4`

`baseString`	`string`	`"Hello"`	6
`derivedBool`	`bool`	`true`	1
`derivedString`	`string`	`"World!"`	7
`derivedDouble`	`double`	`3.14`	8

Member values of an exception of type `Derived`.

From the above table, we can see that the total size of the members of `Base` is 10 bytes, and the total size of the members of `Derived` is 16 bytes. None of the exception members are classes. An instance of this exception has the on-the-wire representation shown in the next table. (The size, type, and byte offset of the marshaled representation is indicated for each component.)

Marshaled value	Size in bytes	Type	Byte offset
`0` *(no class members)*	1	`bool`	0
`"::Derived"` *(type ID)*	10	`string`	1
`20` *(byte count for slice)*	4	`int`	11
`1` (*`derivedBool`*)	1	`bool`	15
`"World!"` (*`derivedString`*)	7	`string`	16
`3.14` (*`derivedDouble`*)	8	`double`	23
`"::Base"` *(type ID)*	7	`string`	31
`14` *(byte count for slice)*	4	`int`	38
`99` (*`baseInt`*)	4	`int`	42
`"Hello"` (*`baseString`*)	6	`string`	46

Marshaled representation of the exception.

Note that the size of each string is one larger than the actual string length. This is because each string is preceded by a count of its number of bytes, as directed by the encoding for strings.

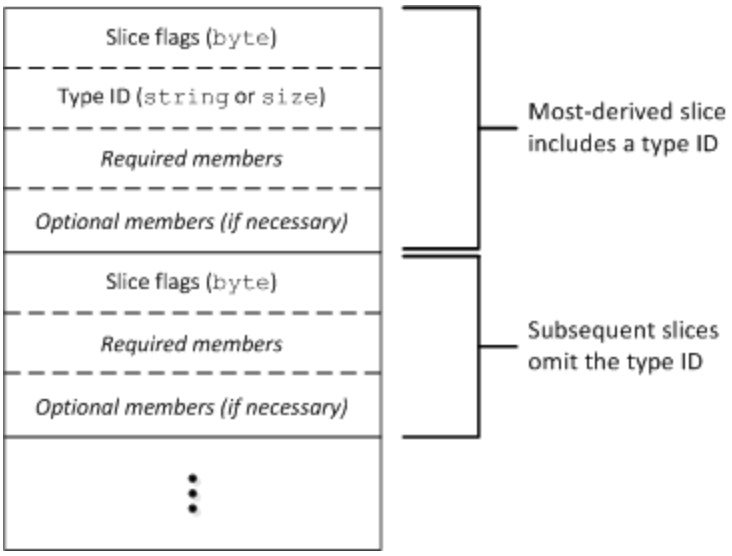
The receiver of this sequence of values uses the header byte to decide whether it eventually must unmarshal any class instances contained in the exception (none in this example) and then examines the first type ID (`::Derived`). If the receiver recognizes that type ID, it can unmarshal the contents of the first slice, followed by the remaining slices; otherwise, the receiver reads the byte count that follows the unknown type (20) and then skips 20-4 bytes in the input stream, which is the start of the type ID for the second slice (`::Base`). If the receiver does not recognize that type ID either, it again reads the byte count following the type ID (14), skips 14-4 bytes, and attempts to read another type ID. (This can happen only if client and server have been compiled with mismatched Slice definitions that disagree in the exception specification of an operation.) In this case, the receiver will eventually encounter an unmarshaling error, which it can report with a `MarshalException`.

If an exception contains class members, these members are marshaled following the exception slices as described in the class encoding.

Exception Encoding version 1.1

An exception is marshaled as a collection of slices whose order matches the inheritance hierarchy, with the most-derived type appearing first. The selected encoding format affects the content of each slice. The final slice, representing the least-derived type, has its *last slice* bit set to true.

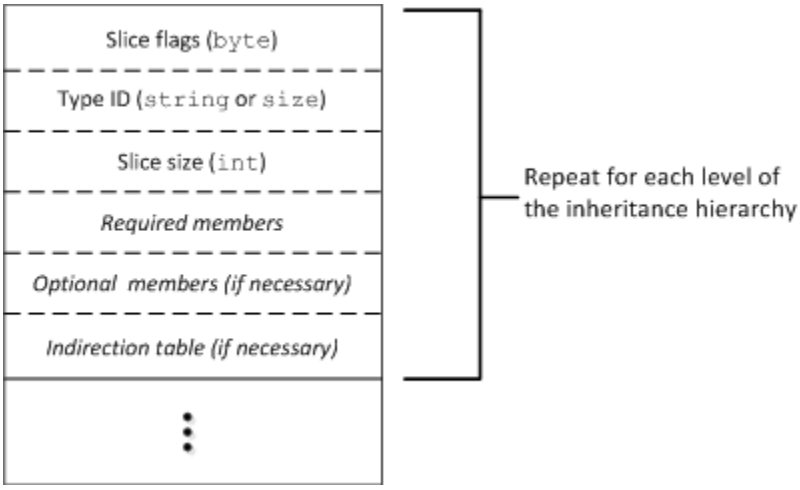
An exception in the compact format is marshaled as follows:



Compact format for exceptions.

The leading byte of each slice is a set of bit flags that specifies the features of the slice. The compact format includes a type ID in the initial (most-derived) slice but omits the type ID from all subsequent slices.

The sliced format includes a type ID in every slice, along with a slice size and an optional indirection table:



Sliced format for exceptions.

To illustrate the marshaling, consider the following exception hierarchy:

Slice

```
exception Base
{
    int baseInt;
    string baseString;
}

exception Derived extends Base
{
    bool derivedBool;
    string derivedString;
    double derivedDouble;
}
```

Assume that the exception members are initialized to the values shown below:

Member	Type	Value	Marshaled size (in bytes)
`baseInt`	`int`	`99`	4
`baseString`	`string`	`"Hello"`	6
`derivedBool`	`bool`	`true`	1
`derivedString`	`string`	`"World!"`	7
`derivedDouble`	`double`	`3.14`	8

Member values of an exception of type `Derived`.

From the above table, we can see that the total size of the members of `Base` is 10 bytes, and the total size of the members of `Derived` is 16 bytes. None of the exception members are classes. An instance of this exception using the sliced format has the on-the-wire representation shown in the next table. (The size, type, and byte offset of the marshaled representation is indicated for each component.)

Marshaled value	Size in bytes	Type	Byte offset
`18` *(flags: type ID is a string, slice size is present)*	1	`byte`	0
`"::Derived"` *(type ID)*	10	`string`	1
`20` *(byte count for slice)*	4	`int`	11
`1` *(`derivedBool`)*	1	`bool`	15
`"World!"` *(`derivedString`)*	7	`string`	16
`3.14` *(`derivedDouble`)*	8	`double`	23
`50` *(flags: type ID is a string, slice size is present, last slice)*	1	`byte`	31
`"::Base"` *(type ID)*	7	`string`	32
`14` *(byte count for slice)*	4	`int`	39
`99` *(`baseInt`)*	4	`int`	43
`"Hello"` *(`baseString`)*	6	`string`	47

Marshaled representation of the exception using the sliced format.

Note that the size of each string is one larger than the actual string length. This is because each string is preceded by a count of its number of bytes, as directed by the encoding for strings.

Repeating this exercise using the compact format produces the following encoding:

Marshaled value	Size in bytes	Type	Byte offset
2 *(flags: type ID is a string)*	1	`byte`	0
`"::Derived"` *(type ID)*	10	`string`	1
1 (*`derivedBool`*)	1	`bool`	11
`"World!"` (*`derivedString`*)	7	`string`	12
`3.14` (*`derivedDouble`*)	8	`double`	19
32 *(flags: last slice)*	1	`byte`	27
99 (*`baseInt`*)	4	`int`	28
`"Hello"` (*`baseString`*)	6	`string`	32

Marshaled representation of the exception using the compact format.

When using the compact format, the receiver *must* know the most-derived type: the only type ID included in the encoding is that of the most-derived type. Furthermore, the lack of slice sizes means the receiver cannot skip a slice without knowing how to decode its contents.

See Also

- Type IDs
- Basic Data Encoding
- Data Encoding for Classes

Data Encoding for Classes

The marshaling for classes is complex, due to the need to deal with the pointer semantics for graphs of classes, as well as the need for the receiver to slice or preserve classes of unknown derived type. In addition, the marshaling for classes uses a type ID compression scheme to avoid repeatedly marshaling the same type IDs for large graphs of class instances.

Encoding for Class References

Consider the following Slice definitions:

Slice

```
class Node
{
    int value;
    Node next;
}

struct S
{
    Node obj;
}
```

We call the `obj` member a *reference* to a class instance. References can appear as data members, as in the case of `obj` above, or nested inside of other types, such as a sequence element or dictionary value. There are significant differences in the encoding for references between versions 1.0 and 1.1.

Class Reference Encoding version 1.0

Ice encodes a nil reference as a 32-bit integer with value zero. For non-nil references, the encoder maintains a table per encapsulation that associates a unique non-zero positive integer with each class instance. We refer to this integer as an *instance ID*. The first time the encoder encounters a reference to a particular instance, it assigns the instance an unused ID, inserts it into the table, and encodes the reference as a 32-bit integer whose value is the *negative* of the ID. If the encoder has already encountered that instance, it simply encodes the instance's previously-assigned ID as a negative 32-bit integer.

All class instances are encoded at the end of the encapsulation.

Class Reference Encoding version 1.1

As in version 1.0 of the encoding, each instance is assigned a unique ID per encapsulation. However, to conserve space, version 1.1 encodes a reference as a size value, with a nil reference encoded as a size value of zero and a non-nil reference encoded as a positive size value. The encoding reserves ID value zero to denote a nil reference, and ID value 1 to denote an *inline* instance in which an instance's state is marshaled at the point of its first reference. Consequently, instance IDs must start at 2. The encoding assigns instance IDs sequentially in order of appearance, with the first instance assigned an ID of 2, the next instance has ID 3, and so on.

The encoding for a reference depends on the format being used, and may also depend on the context in which the reference occurs:

- **Compact format**
  In this format, the encoding always uses the inline scheme. Suppose we are encoding a value of the structure type S shown earlier. If its `obj` member refers to a class instance that the encoder has not yet encountered in the current encapsulation, the encoding assigns it the next available sequential ID, marshals a size with value 1 signifying that an instance is about to be written, and then immediately marshals the instance itself. Any subsequent reference to the same instance is encoded as a size whose value is the instance's corresponding ID.

- **Sliced format**
  When a reference occurs *outside* the context of a class instance, Ice uses the same inline scheme as in the compact format. For example, suppose a Slice operation declared a parameter of structure type S shown above. When marshaling this parameter, its `obj` member is encoded as a size with value 1, followed by the encoding of the `Node` instance itself. Now consider the marshaling of

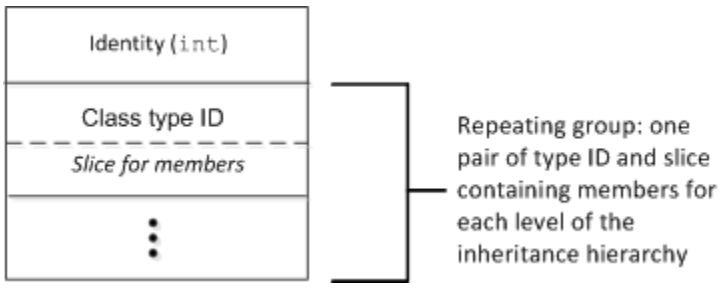
the `Node` instance, where we encounter a class reference in member `next` that occurs *inside* the context of a class instance. To assist the Ice run time in unmarshaling and remarshaling instances, such a reference is encoded as a size whose value is an index into an indirection table.

Encoding for Class Instances

It is important to understand the distinction between marshaling a reference and marshaling an instance. In C++ terms, this is equivalent to the difference between a pointer and the heap data containing an object's state at which the pointer is pointing. Marshaling an instance means we are encoding the data members of a class instance.

Class Instance Encoding version 1.0

Classes are marshaled as a number of pairs containing a type ID and a slice (one pair for each level of the inheritance hierarchy) and marshaled in derived-to-base order. Only data members are marshaled — no information is sent that would relate to operations. Each marshaled class instance is preceded by a (non-zero) positive integer that provides an identity for the instance. The sender assigns this identity during marshaling such that each marshaled instance has a different identity. The receiver uses that identity to correctly reconstruct graphs of classes. The overall marshaling format for classes is shown below:



Marshaling format for classes.

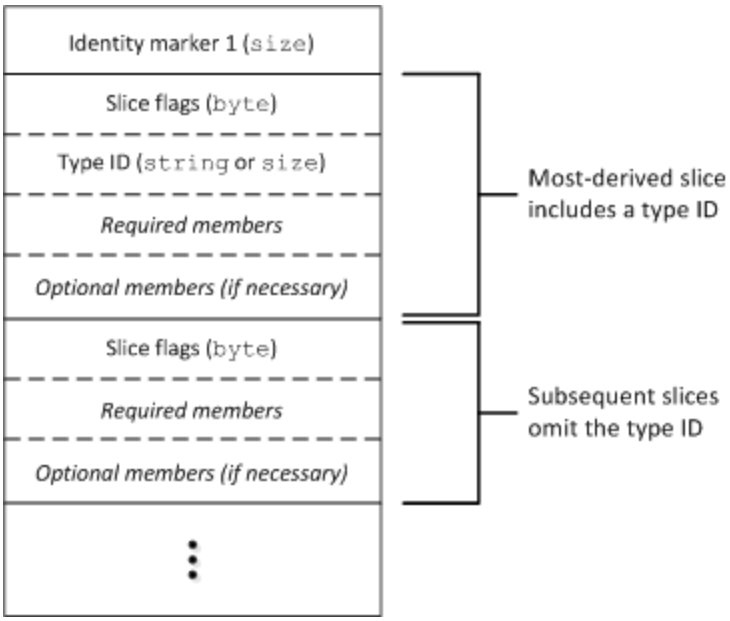
Prior to ending an encapsulation, the sender must encode all instances referenced within that encapsulation in an object table. The encoding may require multiple passes: as the sender processes an instance in the current pass, it may encounter additional instances to be processed in a subsequent pass. For each pass, the sender encodes a size denoting the number of objects in the pass. To signify that all instances have been encoded, the sender must marshal a size value of zero.

This object table must be written if any operation parameter or data member has a class type, even if all references are nil.

Class Instance Encoding version 1.1

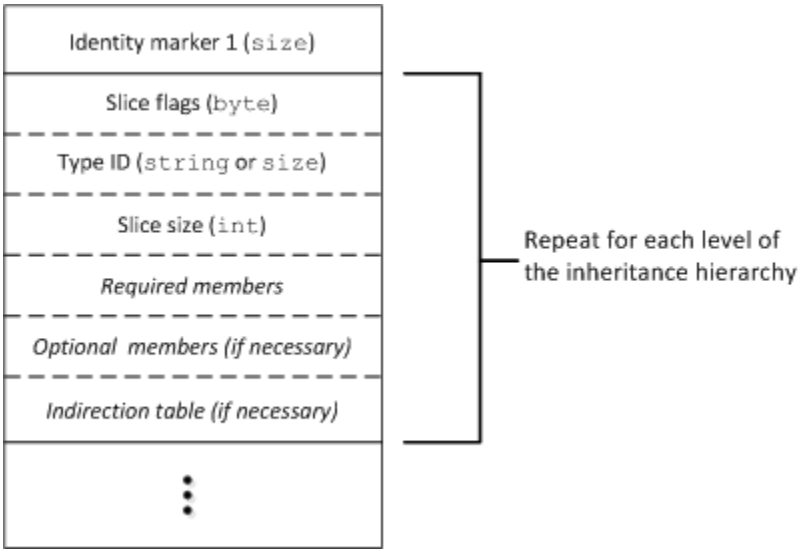
The leading byte of a class instance is a size value of 1. Following this byte is a collection of slices arranged in derived-to-base order. Only data members are marshaled — no information is sent that would relate to operations. The initial (most-derived) slice always includes a type ID that may be encoded as a string or as a numeric value, as specified by the flags that begin each slice. Depending on the format being used, subsequent slices may or may not include a type ID.

Each slice consists of a leading byte representing the slice flags, an optional type ID, an optional slice size, the required members for that slice in order of declaration, and the optional members of that slice. The sender must set the appropriate slice flags to indicate whether the slice includes a type ID, size, and optional data members, and whether this is the last slice of the instance. The compact format only includes a type ID in the initial (most-derived) slice and omits the slice size, as shown in the following diagram:



Compact format for classes.

When using the sliced format, a type ID is included in every slice, along with a slice size:



Sliced format for classes.

A slice that contains at least one non-nil class reference must use an indirection table for all references in that slice. The table is an array of instances encoded using the inline scheme. A leading size value indicates the number of elements in the table; each element is either an instance ID (if the instance has already been encoded within the current encapsulation), or the instance itself denoted by a leading size of 1. The indirection table appears in the encoding immediately following the required and optional data members, *but is not included in the byte count denoted by the slice size*. To skip a slice for an unknown type, the receiver can advance the input stream by the number of bytes specified in the slice size, but then must process all references or instances in the indirection table, if one is present.

To support slice preservation for an instance, the receiver must temporarily retain the slices of any unknown derived types, and also be able to reconstruct the indirection table in its original order for each of these slices in case the instance is later remarshaled.

See Also

- Classes
- Type IDs
- Data Encoding for Exceptions

Data Encoding for Class Type IDs

Type ID Encoding version 1.0

Unlike for exception type IDs, class type IDs are not simple strings. Instead, a class type ID is marshaled as a boolean followed by either a string or a size, to conserve bandwidth. To illustrate this, consider the following class hierarchy:

Slice

```
class Base
{
    // ...
}

class Derived extends Base
{
    // ...
}
```

The type IDs for the class slices are `::Derived` and `::Base`. Suppose the sender marshals three instances of `::Derived` as part of a single request. (For example, two instances could be out-parameters and one instance could be the return value.)

The first instance that is sent on the wire contains the type IDs `::Derived` and `::Base` preceding their respective slices. Because marshaling proceeds in derived-to-base order, the first type ID that is sent is `::Derived`. Every time the sender sends a type ID that it has not sent previously in the same request, it sends the boolean value `false`, followed by the type ID. Internally, the sender also assigns a unique positive number to each type ID. These numbers start at `1` and increment by one for each type ID that has not been marshaled previously. This means that the first type ID is encoded as the boolean value `false`, followed by `::Derived`, and the second type ID is encoded as the boolean value `false`, followed by `::Base`.

When the sender marshals the remaining two instances, it consults a lookup table of previously-marshaled type IDs. Because both type IDs were sent previously in the same request (or reply), the sender encodes all further occurrences of `::Derived` as the value `true` followed by the number `1` encoded as a size, and it encodes all further occurrences of `::Base` as the value `true` followed by the number `2` encoded as a size.

When the receiver reads a type ID, it first reads its boolean marker:

- If the boolean is `false`, the receiver reads a string and enters that string into a lookup table that maps integers to strings. The first new class type ID received in a request is numbered `1`, the second new class type ID is numbered `2`, and so on.
- If the boolean value is `true`, the receiver reads a number encoded as a size and uses it to retrieve the corresponding class type ID from the lookup table.

Note that this numbering scheme is re-established for each new encapsulation. (As we will see in our discussion of protocol messages, parameters, return values, and exceptions are always marshaled inside an enclosing encapsulation.) For subsequent or nested encapsulation, the numbering scheme restarts, with the first new type ID being assigned the value `1`. In other words, each encapsulation uses its own independent numbering scheme for class type IDs to satisfy the constraint that encapsulations must not depend on their surrounding context.

Encoding class type IDs in this way provides significant savings in bandwidth: whenever an ID is marshaled a second and subsequent time, it is marshaled as a two-byte value (assuming no more than 254 distinct type IDs per request) instead of as a string. Because type IDs can be long, especially if you are using nested modules, the savings are considerable.

Type ID Encoding version 1.1

Each slice of a class instance has a leading byte containing flags that describe various aspects of the slice, including whether the slice includes a type ID and how that type ID is encoded. There are four possibilities:

1. No type ID included
2. Type ID is encoded as a string
3. Type ID is encoded as an index
4. Type ID is encoded as a compact ID

The initial slice of an instance, representing the most-derived type, always contains some form of type ID so that the receiver knows what Slice type is present. Depending on the format used by the sender, subsequent slices may or may not include a type ID: the compact format omits type IDs in subsequent slices, whereas the sliced format includes a type ID in every slice. A receiver need only examine the slice flags to discover how to decode the type ID.

String Type IDs

The encoding for string type IDs uses a "compression" scheme similar to that of version 1.0: within an encapsulation, a given type ID is never encoded as a string more than once. The first time a sender encounters a type ID, the sender assigns an integer index to the ID and encodes the ID as a string. For all subsequent occurrences of the same type ID within the encapsulation, the sender encodes the index associated with that type ID as a size. Index values start at 1 and increase sequentially with each new type ID. The sender is responsible for setting the relevant bits in the flags of each slice to specify how the type ID is encoded.

The slice flags in version 1.1 of the encoding serve the same purpose as the boolean value that precedes each type ID in version 1.0, without consuming an entire byte.

Compact Type IDs

As a way of further reducing the overhead associated with class instances, version 1.1 adds the ability to substitute numeric values for strings when encoding type IDs. Consider the following Slice definitions:

Slice

```
class Base(3)
{
    // ...
}

class Derived(4) extends Base
{
    // ...
}

class MoreDerived extends Derived
{
    // ...
}
```

The value in parentheses after the class name represents the *compact* type ID for the class. The sender's format determines whether each slice includes a type ID. If a given slice includes a type ID, the encoding always uses a compact type ID (if defined) in preference to its string equivalent. Suppose a sender is encoding an instance of `MoreDerived` in the sliced format. The initial slice uses the string type ID for `MoreDerived`, the next slice uses the compact type ID for `Derived`, and the last slice uses the compact type ID for `Base`.

A compact type ID is encoded as a size, with the relevant bits set in the slice flags.

> The developer is responsible for ensuring compact type IDs are sufficiently unique. Using values less than 255 produces the most efficient encoding.

See Also

- Type IDs
- Basic Data Encoding
- Protocol Messages

Simple Example of Class Encoding

On this page:

Sample Class Definitions

We have separately discussed the primary components of the class encoding: slices, references, and type IDs. To make the preceding discussions more concrete, consider the following class definitions:

Slice

```
class Base
{
    int baseInt;
    string baseString;
}

class Derived extends Base
{
    bool derivedBool;
    string derivedString;
    double derivedDouble;
}
```

Suppose the sender marshals two instances of `Derived` (for example, as two in-parameters in the same request) with these member values:

First instance:

Member	Type	Value	Marshaled size (in bytes)
`baseInt`	`int`	`99`	4
`baseString`	`string`	`"Hello"`	6
`derivedBool`	`bool`	`true`	1
`derivedString`	`string`	`"World!"`	7
`derivedDouble`	`double`	`3.14`	8

Second instance:

Member	Type	Value	Marshaled size (in bytes)
`baseInt`	`int`	`115`	4
`baseString`	`string`	`"Cave"`	5
`derivedBool`	`bool`	`false`	1
`derivedString`	`string`	`"Canem"`	6
`derivedDouble`	`double`	`6.32`	8

We describe how to marshal these instances using versions 1.0 and 1.1 of the encoding in separate sections below.

Class Encoding version 1.0

The sender arbitrarily assigns a non-zero identity to each instance. Typically, the sender will simply consecutively number the instances starting at `1`. For this example, assume that the two instances have the identities `1` and `2`. The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) is shown below:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(identity)*	4	`int`	0
`0` *(marker for class type ID)*	1	`bool`	4
`"::Derived"` *(class type ID)*	10	`string`	5
`20` *(byte count for slice)*	4	`int`	15
`1` (*`derivedBool`*)	1	`bool`	19
`"World!"` (*`derivedString`*)	7	`string`	20
`3.14` (*`derivedDouble`*)	8	`double`	27
`0` *(marker for class type ID)*	1	`bool`	35
`"::Base"` *(type ID)*	7	`string`	36
`14` *(byte count for slice)*	4	`int`	43
`99` (*`baseInt`*)	4	`int`	47
`"Hello"` (*`baseString`*)	6	`string`	51
`0` *(marker for class type ID)*	1	`bool`	57
`"::Ice::Object"` *(class type ID)*	14	`string`	58
`5` *(byte count for slice)*	4	`int`	72
`0` *(number of dictionary entries)*	1	`size`	76
`2` *(identity)*	4	`int`	77
`1` *(marker for class type ID)*	1	`bool`	81
`1` *(class type ID)*	1	`size`	82
`19` *(byte count for slice)*	4	`int`	83
`0` (*`derivedBool`*)	1	`bool`	87
`"Canem"` (*`derivedString`*)	6	`string`	88
`6.32` (*`derivedDouble`*)	8	`double`	94
`1` *(marker for class type ID)*	1	`bool`	102
`2` *(class type ID)*	1	`size`	103
`13` *(byte count for slice)*	4	`int`	104
`115` (*`baseInt`*)	4	`int`	108
`"Cave"` (*`baseString`*)	5	`string`	112
`1` *(marker for class type ID)*	1	`bool`	117
`3` *(class type ID)*	1	`size`	118
`5` *(byte count for slice)*	4	`int`	119
`0` *(number of dictionary entries)*	1	`size`	123

Note that, because classes (like exceptions) are sent as a sequence of slices, the receiver of a class can slice off any derived parts of a class it does not understand. Also note that (as shown in the above table) each class instance contains three slices. The third slice is for the type `::Ice::Object`, which is the base type of all classes. The class type ID `::Ice::Object` has the number `3` in this example because

it is the third distinct type ID that is marshaled by the sender. (See entries at byte offsets 58 and 118 in the above table.) All class instances have this final slice of type `::Ice::Object`.

Marshaling a separate slice for `::Ice::Object` dates back to Ice versions 1.3 and earlier. In those versions, classes carried a facet map that was marshaled as if it were defined as follows:

Slice

```
module Ice
{
    class Object;

    dictionary<string, Object> FacetMap;

    class Object
    {
        FacetMap facets; // No longer exists
    }
}
```

As of Ice version 1.4, this facet map is always empty, that is, the count of entries for the dictionary that is marshaled in the `::Ice::Object` slice is always zero. If a receiver receives a class instance with a non-empty facet map, it must throw a `MarshalException`.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Class Encoding version 1.1

A leading size value of `1` marks the beginning of an instance, followed by one or more slices.

Class Encoding in the Sliced Format

The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) in the sliced format is shown below:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(instance marker)*	1	`size`	0
`17` *(slice flags: string type ID, size is present)*	1	`byte`	1
`"::Derived"` *(type ID - assigned index 1)*	10	`string`	2
`20` *(byte count for slice)*	4	`int`	12
`1` *(`derivedBool`)*	1	`bool`	16
`"World!"` *(`derivedString`)*	7	`string`	17
`3.14` *(`derivedDouble`)*	8	`double`	24
`49` *(slice flags: string type ID, size is present, last slice)*	1	`byte`	32
`"::Base"` *(type ID - assigned index 2)*	7	`string`	33
`14` *(byte count for slice)*	4	`int`	40
`99` *(`baseInt`)*	4	`int`	44
`"Hello"` *(`baseString`)*	6	`string`	48
`1` *(instance marker)*	1	`size`	54
`18` *(slice flags: index type ID, size is present)*	1	`byte`	55

`1` *(type ID index for `Derived`)*	`1`	`size`	56
`19` *(byte count for slice)*	`4`	`int`	57
`0` *(`derivedBool`)*	`1`	`bool`	61
`"Canem"` *(`derivedString`)*	6	`string`	62
`6.32` *(`derivedDouble`)*	8	`double`	68
`50` *(slice flags: index type ID, size is present, last slice)*	`1`	`byte`	76
`2` *(type ID index for `Base`)*	`1`	`size`	77
`13` *(byte count for slice)*	`4`	`int`	78
`115` *(`baseInt`)*	`4`	`int`	82
`"Cave"` *(`baseString`)*	5	`string`	86

The sliced format allows the receiver of a class to slice off any derived parts of a class it does not understand, as in version 1.0 of the encoding. Although the sliced format provides equivalent functionality to that of version 1.0, it is significantly more efficient, requiring only 91 bytes to encode our example compared to the 124 bytes required by version 1.0. We could reduce the encoded size even further, while still retaining the ability to slice off unknown types, by using compact type IDs.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Class Encoding in the Compact Format

The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) in the compact format is shown below:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(instance marker)*	`1`	`size`	0
`1` *(slice flags: string type ID)*	`1`	`byte`	1
`"::Derived"` *(type ID - assigned index `1`)*	10	`string`	2
`1` *(`derivedBool`)*	`1`	`bool`	12
`"World!"` *(`derivedString`)*	7	`string`	13
`3.14` *(`derivedDouble`)*	8	`double`	20
`32` *(slice flags: last slice)*	`1`	`byte`	28
`99` *(`baseInt`)*	`4`	`int`	29
`"Hello"` *(`baseString`)*	6	`string`	33
`1` *(instance marker)*	`1`	`size`	39
`2` *(slice flags: index type ID)*	`1`	`byte`	40
`1` *(type ID index for `Derived`)*	`1`	`size`	41
`0` *(`derivedBool`)*	`1`	`bool`	42
`"Canem"` *(`derivedString`)*	6	`string`	43
`6.32` *(`derivedDouble`)*	8	`double`	49
`32` *(slice flags: last slice)*	`1`	`byte`	57
`115` *(`baseInt`)*	`4`	`int`	58
`"Cave"` *(`baseString`)*	5	`string`	62

In an effort to conserve bandwidth, the compact format omits certain details that would allow a receiver to slice off derived parts of a class, such as the slice size and the type IDs for base classes. The result is an encoding that requires only 67 bytes for the two sample instances.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Class Encoding in the Compact Format with Compact Type IDs

Compact type IDs can be used regardless of the sender's chosen format. For the sake of example, we will use compact type IDs together with the compact format to produce the smallest encoding possible. The Slice definitions below reflect the addition of the compact type IDs:

Slice

```
class Base(10)
{
    int baseInt;
    string baseString;
}

class Derived(11) extends Base
{
    bool derivedBool;
    string derivedString;
    double derivedDouble;
}
```

We assign the compact type ID `10` to `Base` and `11` to `Derived`. Note however that assigning a compact type ID to `Base` does not affect the size of the encoded data in our example because the compact format omits type IDs altogether for base types.

The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) in the compact format is shown below:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(instance marker)*	1	`size`	0
`3` *(slice flags: compact type ID)*	1	`byte`	1
`11` *(compact type ID for `Derived`)*	1	`size`	2
`1` *(`derivedBool`)*	1	`bool`	3
`"World!"` *(`derivedString`)*	7	`string`	4
`3.14` *(`derivedDouble`)*	8	`double`	11
`32` *(slice flags: last slice)*	1	`byte`	19
`99` *(`baseInt`)*	4	`int`	20
`"Hello"` *(`baseString`)*	6	`string`	24
`1` *(instance marker)*	1	`size`	30
`3` *(slice flags: compact type ID)*	1	`byte`	31
`11` *(compact type ID for `Derived`)*	1	`size`	32
`0` *(`derivedBool`)*	1	`bool`	33
`"Canem"` *(`derivedString`)*	6	`string`	34
`6.32` *(`derivedDouble`)*	8	`double`	40
`32` *(slice flags: last slice)*	1	`byte`	48
`115` *(`baseInt`)*	4	`int`	49
`"Cave"` *(`baseString`)*	5	`string`	53

Substituting a compact type ID for its string equivalent reduces the encoded size for the two instances by another nine bytes to 58, less than half the size of version 1.0.

See Also

- Data Encoding for Classes
- Data Encoding for Exceptions
- Basic Data Encoding
- Type IDs

Data Encoding for Class Graphs

Classes support pointer semantics, that is, you can construct graphs of classes. It follows that classes can arbitrarily point at each other, and therefore the encoding must provide a scheme for serializing and deserializing a class graph. Note that the marshaling format for class references and instances differs significantly between versions 1.0 and 1.1 of the encoding.

On this page:

Encoding a Class Graph version 1.0

In version 1.0 of the encoding, an instance ID is used to distinguish instances and pointers as follows:

- An instance ID of 0 denotes a null pointer.
- An instance ID > 0 precedes the marshaled contents of an instance.
- An instance ID < 0 denotes a pointer to an instance.

Instance ID values less than zero are pointers. For example, if the receiver receives the instance ID `-57`, this means that the corresponding class member that is currently being unmarshaled will eventually point at the instance with ID `57`.

For structures, classes, exceptions, sequences, and dictionary members that do not contain class members, the Ice encoding uses a simple depth-first traversal algorithm to marshal the members. For example, structure members are marshaled in the order of their Slice definition; if a structure member itself is of complex type, such as a sequence, the sequence is marshaled in toto where it appears inside its enclosing structure. For complex types that contain class members, this depth-first marshaling is suspended: instead of marshaling the actual class instance at this point, a negative instance ID is marshaled that indicates which class instance that member must eventually denote. For example, consider the following definitions:

Slice

```
class C
{
    // ...
}

struct S
{
    int i;
    C firstC;
    C secondC;
    C thirdC;
    int j;
}
```

Suppose we initialize a structure of type S as follows:

C++11 C++98

```
S myS;
myS.i = 99;
myS.firstC = make_shared<C>();  // New instance
myS.secondC = nullptr;          // null
myS.thirdC = myS.firstC;        // Same instance as previously
myS.j = 100;
```

```
S myS;
myS.i = 99;
myS.firstC = new C;             // New instance
myS.secondC = 0;                // null
myS.thirdC = myS.firstC;        // Same instance as previously
myS.j = 100;
```

When this structure is marshaled, the contents of the three class members are not marshaled in-line. Instead, the sender marshals the negative instance IDs of the corresponding instances. Assuming that the sender has assigned the instance ID `78` to the instance assigned to `myS.firstC`, `myS` is marshaled as shown in the table.

Marshaled Value	Size in Bytes	Type	Byte offset
`99` *(myS.i)*	`4`	`int`	`0`
`-78` *(myS.firstC)*	`4`	`int`	`4`
`0` *(myS.secondC)*	`4`	`int`	`8`
`-78` *(mys.thirdC)*	`4`	`int`	`12`
`100` *(myS.j)*	`4`	`int`	`16`

Note that `myS.firstC` and `myS.thirdC` both use the instance ID `-78`. This allows the receiver to recognize that `firstC` and `thirdC` poin t at the same class instance (rather than at two different instances that happen to have the same contents).

Marshaling the negative instance IDs instead of the contents of an instance allows the receiver to accurately reconstruct the class graph that was sent by the sender. However, this begs the question of *when* the actual instances are to be marshaled as described at the beginning of this section. In Ice protocol messages, parameters and return values are marshaled as if they were members of a structure. For example, if an operation invocation has five input parameters, the client marshals the five parameters end-to-end as if they were members of a single structure. If any of the five parameters are class instances, or are of complex type (recursively) containing class instances, the sender marshals the parameters in multiple passes: the first pass marshals the parameters end-to-end, using the usual depth-first algorithm:

- If the sender encounters a class member during marshaling, it checks whether it has marshaled the same instance previously for the current request or reply:
  - If the instance has not been marshaled before, the sender assigns a new instance ID to the instance and marshals the negative ID.
  - Otherwise, if the instance was marshaled previously, the sender sends the same negative ID that it previously sent for that instance.

In effect, during marshaling, the sender builds a table that is indexed by the address of each instance; the lookup value for the instance is its ID.

Once the first pass ends, the sender has marshaled all the parameters, but has not yet marshaled any of the class instances that may be pointed at by various parameters or members. The instance table at this point contains all those instances for which negative IDs (pointers) were marshaled, so whatever is in the table at this point are the classes that the receiver still needs. The sender now marshals those instances in the table, but with positive IDs and followed by their contents, as described in our earlier example. The outstanding instances are marshaled as a sequence, that is, the sender marshals the number of instances as a size, followed by the actual instances.

In turn, the instances just sent may themselves contain class members; when those class members are marshaled, the sender assigns IDs to new instances or uses a negative ID for previously marshaled instances as usual. This means that, by the end of the second pass, the table may have grown, necessitating a third pass. That third pass again marshals the outstanding class instances as a size followed by the actual instances. The third pass contains all those instances that were not marshaled in the second pass. Of course, the third pass may

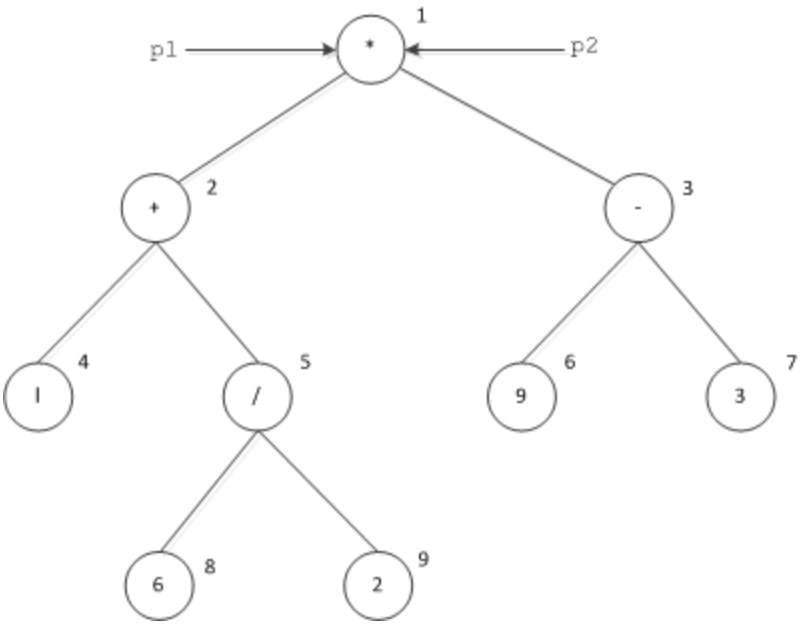
trigger yet more passes until, finally, the sender has sent all outstanding instances, that is, marshaling is complete. At this point, the sender terminates the sequence of passes by marshaling an empty sequence (the value `0` encoded as a size).

To illustrate this with an example, consider the definitions shown in Classes with Operations once more:

Slice

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not }
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or }

class Node
{
    idempotent long eval();
}

class UnaryOperator extends Node
{
    UnaryOp operator;
    Node operand;
}

class BinaryOperator extends Node
{
    BinaryOp op;
    Node operand1;
    Node operand2;
}

class Operand
{
    long val;
}
```

These definitions allow us to construct expression trees. Suppose the client initializes a tree to the shape shown in the illustration below, representing the expression `(1 + 6 / 2) * (9 - 3)`. The values outside the nodes are the identities assigned by the client.



Expression tree for the expression `(1 + 6 / 2) * (9 - 3)`. Both p1 and p2 denote the root node.

The client passes the root of the tree to the following operation in the parameters `p1` and `p2`, as shown in the illustration above. (Even though it does not make sense to pass the same parameter value twice, we do it here for illustration purposes):

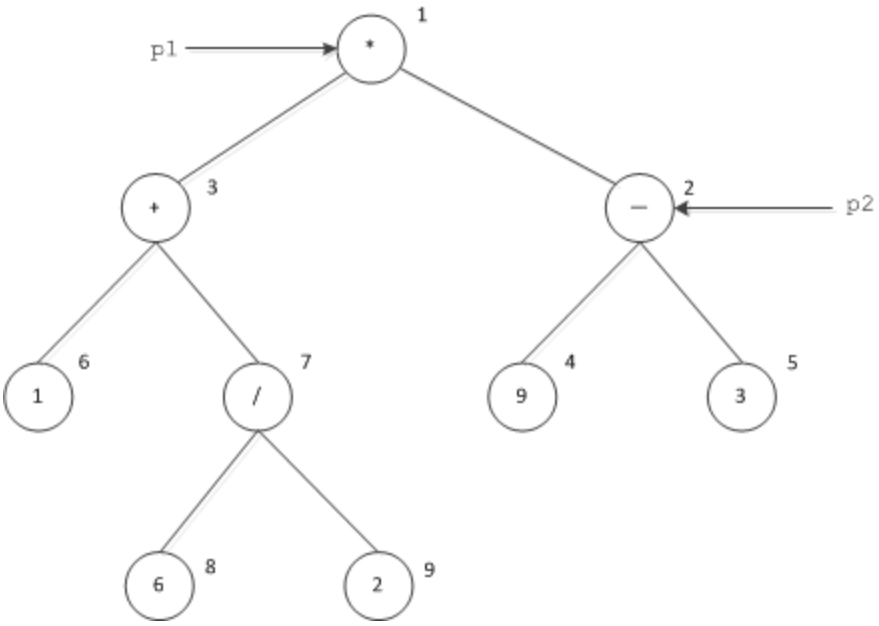
Slice

```
interface Tree
{
    void sendTree(Node p1, Node p2);
}
```

The client now marshals the two parameters `p1` and `p2` to the server, resulting in the value `-1` being sent twice in succession. (The client arbitrarily assigns an instance ID to each node. The value of the ID does not matter, as long as each node has a unique ID. For simplicity, the Ice implementation numbers instances with a counter that starts counting at `1` and increments by one for each unique instance.) This completes the marshaling of the parameters and results in a single instance with ID 1 in the instance table. The client now marshals a sequence containing a single element, node 1, as described in the example. In turn, node 1 results in nodes 2 and 3 being added to the table, so the next sequence of nodes contains two elements, nodes 2 and 3. The next sequence of nodes contains nodes 4, 5, 6, and 7, followed by another sequence containing nodes 8 and 9. At this point, no more class instances are outstanding, and the client marshals an empty sequence to indicate to the receiver that the final sequence has been marshaled.

Within each sequence, the order in which class instances are marshaled is irrelevant. For example, the third sequence could equally contain nodes 7, 6, 4, and 5, in that order. What is important here is that each sequence contains nodes that are an equal number of "hops" away from the initial node: the first sequence contains the initial node(s), the second sequence contains all nodes that can be reached by traversing a single link from the initial node(s), the third sequence contains all nodes that can be reached by traversing two links from the initial node(s), and so on.

Now consider the same example once more, but with different parameter values for `sendTree`: `p1` denotes the root of the tree, and `p2` deno tes the `-` operator of the right-hand sub-tree, as shown below:



The expression tree of with p1 and p2 denoting different nodes.

The graph that is marshaled is exactly the same, but instances are marshaled in a different order and with different IDs:

- During the first pass, the client sends the IDs `-1` and `-2` for the parameter values.
- The second pass marshals a sequence containing nodes 1 and 2.
- The third pass marshals a sequence containing nodes 3, 4, and 5.
- The fourth pass marshals a sequence containing nodes 6 and 7.
- The fifth pass marshals a sequence containing nodes 8 and 9.
- The final pass marshals an empty sequence.

In this way, any graph of nodes can be transmitted (including graphs that contain cycles). The receiver reconstructs the graph by filling in a patch table during unmarshaling:

- Whenever the receiver unmarshals a negative ID, it adds that ID to a patch table; the lookup value is the memory address of the parameter or member that eventually will point at the corresponding instance.
- Whenever the receiver unmarshals an actual instance, it adds the instance to an unmarshaled table; the lookup value is the memory address of the instantiated class. The receiver then uses the address of the instance to patch any parameters or members with the actual memory address.

Note that the receiver may receive negative IDs that denote class instances that have been unmarshaled already (that is, point "backward" in the unmarshaling stream), as well as instances that are yet to be unmarshaled (that is, point "forward" in the unmarshaling stream). Both scenarios are possible, depending on the order in which instances are marshaled, as well as their in-degree.

To provide another example, consider the following definition:

Slice

```
class C
{
    // ...
}

sequence<C> CSeq;
```

Suppose the client marshals a sequence of 100 `C` instances to the server, with each instance being distinct. (That is, the sequence contains 100 pointers to 100 different instances, not 100 pointers to the same single instance.) In that case, the sequence is marshaled as a size of 100, followed by 100 negative IDs, `-1` to `-100`. Following that, the client marshals a single sequence containing the 100 instances, each instance with its positive ID in the range `1` to `100`, and completes by marshaling an empty sequence.

On the other hand, if the client sends a sequence of 100 elements that all point to the same single class instance, the client marshals the sequence as a size of 100, followed by 100 negative IDs, all with the value `-1`. The client then marshals a sequence containing a single element, namely instance `1`, and completes by marshaling an empty sequence.

Encoding a Class Graph version 1.1

The most significant difference in the class encoding between version 1.0 and 1.1 is the location of class instances in the output stream. In version 1.0, instances are always marshaled at the end of the encapsulation, whereas in version 1.1 it is possible for instances to be marshaled *inline* at the point of first reference. Our previous discussion of class references describes the factors that determine whether a given class reference is marshaled as a reference or as an inline instance. The encoding rules can be summarized as follows:

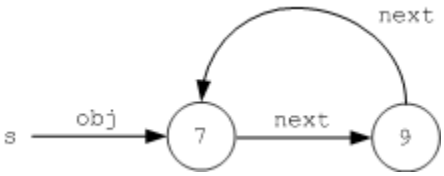
- When using the compact format, always marshal an instance inline at the point of its first reference, and marshal all subsequent occurrences of the same instance as a reference.
- When using the sliced format, the encoding depends on the context in which a reference occurs: if the reference occurs while encoding a slice of an object or exception, encode it as an index into an indirection table that appears at the end of the slice, otherwise encode it as an inline instance or reference.

We can use the Slice definitions below to explore these situations:

Slice

```
class Node
{
    int value;
    Node next;
}

struct S
{
    Node obj;
}
```

Suppose we create an instance of structure `S` and assign it to the variable `s`, then construct the following class graph:



Class graph with circular reference.

We discuss the encoding for these values in separate sections below.

Class Graph in the Compact Format

As we marshal the structure's `obj` member, the compact format produces the following encoding:

Marshaled value	Size in bytes	Type	Byte offset
`1` (*`obj` - inline instance marker - assigned ID 2*)	`1`	`size`	`0`
`33` (*slice flags: string type ID, last slice*)	`1`	`byte`	`1`
`"::Node"` (*type ID - assigned index 1*)	`7`	`string`	`2`
`7` (*`value`*)	`4`	`int`	`9`
`1` (*`next` - inline instance marker - assigned ID 3*)	`1`	`size`	`13`

34 (*slice flags: type ID index, last slice*)	1	byte	14
1 (*type ID index*)	1	size	15
9 (*value*)	4	int	16
2 (*next - reference to ID 2*)	1	size	20

This example shows that inline instances are marshaled immediately upon discovery. As we encode the `obj` member, we encounter a new `Node` instance, assign it the ID 2, and begin to encode it inline using the reserved instance marker value `1`. Next, we encode the instance's `value` member and then examine its `next` member. Since we have not encountered this instance yet, we assign it the ID 3 and begin to encode this new instance inline. Its `next` member is a circular reference to the object with ID 2, so we simply encode a reference to this instance to complete the structure.

Class Graph in the Sliced Format

The need to support both slicing and slice preservation makes the sliced format more complex than the compact format. The encoding for our example values shows how class instances are handled differently depending on their context:

Marshaled value	Size in bytes	Type	Byte offset
1 (*obj - inline instance marker - assigned ID 2*)	1	size	0
57 (*slice flags: string type ID, size is present, indirection table is present, last slice*)	1	byte	1
`"::Node"` (*type ID - assigned index 1*)	7	string	2
9 (*byte count for slice*)	4	int	9
7 (*value*)	4	int	13
1 (*next - indirection table index*)	1	size	17
1 (*size of indirection table*)	1	size	18
1 (*indirection table entry - inline instance marker - assigned ID 3*)	1	size	19
58 (*slice flags: type ID index, size is present, indirection table is present, last slice*)	1	byte	20
1 (*type ID index*)	1	size	21
9 (*byte count for slice*)	4	int	22
9 (*value*)	4	int	26
1 (*next - indirection table index - ID 2*)	1	size	30
1 (*size of indirection table*)	1	size	31
2 (*indirection table entry - reference to ID 2*)	1	size	32

For this example, let us assume that the structure value is an operation parameter, and therefore the structure member `obj` occurs *outside* the context of a class or exception slice. As a result, we encode the instance with ID 2 as an inline instance. However, when we process the instance's `next` member, we are now *inside* the context of a class slice, so we must use an indirection table. The sender adds an entry to the indirection table for each unique class reference in the slice; instead of encoding a reference or inline instance, the sender encodes the index of the corresponding entry in the table. As shown above, the value encoded for the `next` member is `1`, representing the first entry in the table (`0` is still reserved for nil references).

The indirection table follows immediately after the last data member in the slice, but the table is *not* included in the byte count for the slice. A leading size denotes the number of entries in the table. The one and only entry in this table, the instance with ID 3, has not yet been encoded, therefore it is marshaled immediately as an inline instance. This instance also uses an indirection table, although in this case the table entry is simply a reference to instance ID 2, which has already been encoded.

Importance of the Indirection Table

The indirection table is necessary for implementing the slice preservation feature. Normally, when a receiver does not recognize the type ID in a slice, it has the option of ignoring that slice by skipping ahead in the stream by the number of bytes in the slice. However, when slice preservation is enabled, the receiver must keep a copy of the slice data in case the instance is later remarshaled. The need for the

indirection table becomes apparent when you consider that an opaque blob of slice data may contain class references, and those class references can change during remarshaling. For example, without an indirection table, the sender might encode the instance ID 3 as the value of member `next`, but what happens if the receiver assigns that instance a different ID, such as 12, when it remarshals the preserved slice? The receiver preserved the slice because it did not understand the type ID, which means it does not know the contents of the slice data and therefore it cannot "patch" any class references the slice might contain. The indirection table serves as an external "patch table" to solve this problem, essentially making the opaque slice data *relocatable* with respect to class references.

The byte count for a slice does not include the indirection table because the receiver must process the table regardless of whether it recognizes that slice's type ID. Consequently, to "skip" a slice, the receiver can skip (or preserve) the number of bytes specified by the slice's byte count, but still must decode the indirection table if the slice flags indicate that a table is present. If the receiver preserves the slice, it must also associate an indirection table with that slice; during remarshaling, the sender copies the opaque slice data into the stream, and then reconstructs the indirection table using (potentially) new instance IDs for the instances referenced in the table.

It is possible that the *only* reference to an instance is in an indirection table. To properly implement slice preservation, a receiver must therefore retain *every* instance that is referenced by a preserved indirection table. This is true even if the receiver does not recognize any of the type IDs in an instance; in effect, the receiver must construct a temporary "unknown object" placeholder for the instance, whose only purpose is to encapsulate the data comprising its slices in case the instance is later remarshaled.

Impact of Slicing on Class Graph Decoding

It is important to note that when a graph of class instances is sent, it always forms a connected graph. However, when the receiver rebuilds the graph, it may end up with a disconnected graph, due to slicing. Consider:

Slice

```
class Base
{
    // ...
}

class Derived extends Base
{
    // ...
    Base b;
}

interface Example
{
    void op(Base p);
}
```

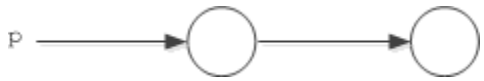
Suppose the client has complete type knowledge, that is, understands both types `Base` and `Derived`, but the server only understands type `Base`, so the derived part of a `Derived` instance is sliced. The client can instantiate classes to be sent as parameter `p` as follows:

C++11 C++98

```
auto p = make_shared<Derived>();
p->b = make_shared<Derived>();
shared_ptr<ExamplePrx> e = ...;
e->op(p);
```

```
DerivedPtr p = new Derived;
p->b = new Derived;
ExamplePrx e = ...;
e->op(p);
```

As far as the client is concerned, the graph looks like the one shown below:



Sender-side view of a graph containing derived instances.

However, the server does not understand the derived part of the instances and slices them. Yet, the server unmarshals all the class instances, leading to the situation where the class graph has become disconnected, as shown here:

Receiver-side view of the graph.

Of course, more complex situations are possible, such that the receiver ends up with multiple disconnected graphs, each containing many instances.

> The slice preservation feature in version 1.1 of the encoding allows a receiver to remarshal the original graph intact, despite the fact that the receiver's in-memory object graph may appear to be disconnected.

See Also

- Classes with Operations
- Basic Data Encoding
- Data Encoding for Classes
- Simple Example of Class Encoding
- Protocol Messages
- Slicing Values and Exceptions

Data Encoding for Interfaces

Interfaces can be marshaled by value. For an interface marshaled by value (as opposed to a class instance derived from that interface), only the type ID of the most-derived interface is encoded. We use the following Slice definitions in our discussion:

Slice

```
interface Base { /* ... */ }

interface Derived extends Base { /* ... */ }

interface Example
{
    void doSomething(Base b);
}
```

We assume that the client passes a class instance to `doSomething` that does not have a Slice definition (but implements `Derived`).

Interface Encoding version 1.0

The on-the-wire representation of the interface parameter using encoding version 1.0 is as follows:

Marshaled Value	Size in Bytes	Type	Byte offset
`1` *(identity)*	4	`int`	0
`0` *(marker for class type ID)*	1	`bool`	4
`"::Derived"` *(class type ID)*	10	`string`	5
`4` *(byte count for slice)*	4	`int`	15
`0` *(marker for class type ID)*	1	`bool`	19
`"::Ice::Object"` *(class type ID)*	14	`string`	20
`5` *(byte count for slice)*	4	`int`	34
`0` *(number of dictionary entries)*	1	`size`	38

Interface Encoding version 1.1

With encoding version 1.1, the interface parameter is encoded as if it was a class with no data members. Here is the on-the-wire representation using the sliced format:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(instance marker)*	1	`size`	0
`17` *(slice flags: string type ID, size is present)*	1	`byte`	1
`"::Derived"` *(type ID - assigned index 1)*	10	`string`	2
`4` *(byte count for slice)*	4	`int`	12

The compact format omits the byte count for the slice:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(instance marker)*	1	`size`	0
`1` *(slice flags: string type ID)*	1	`byte`	1

`"::Derived"` *(type ID - assigned index 1)*	10	`string`	2

See Also

- Passing Interfaces by Value
- Type IDs
- Data Encoding for Classes

Data Encoding for Proxies

On this page:

Encoding for Proxy Options

The encoding format of proxies changed in version 1.1.

Proxy Encoding version 1.0

The first component of an encoded proxy is a value of type `Ice::Identity`. If the proxy is a nil value, the `category` and `name` members are empty strings, and no additional data is encoded. The encoding for a non-null proxy consists of proxy options followed by endpoints.

The proxy options are encoded as if they were members of the following structure:

Slice

```
struct ProxyData
{
    Ice::Identity id;
    Ice::StringSeq facet;
    byte mode;
    bool secure;
}
```

The proxy options are described in the table below:

Option	Description
`id`	The object identity
`facet`	The facet name (zero- or one-element sequence)
`mode`	The proxy mode (`0`=twoway, `1`=oneway, `2`=batch oneway, `3`=datagram, `4`=batch datagram)
`secure`	`true` if secure endpoints are required, otherwise `false`

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a proxy with a `facet` field with more than one element, it must throw a `ProxyUnmarshalException`.

Proxy Encoding version 1.1

Version 1.1 of the encoding adds two options to the existing proxy options in version 1.0: protocol and encoding versions. The proxy options are encoded as if they were members of the following structure:

Slice

```
struct ProtocolVersion
{
    byte major;
    byte minor;
}

struct EncodingVersion
{
    byte major;
    byte minor;
}
struct ProxyData
{
    Ice::Identity id;
    Ice::StringSeq facet;
    byte mode;
    bool secure;
    ProtocolVersion protocol;
    EncodingVersion encoding;
}
```

The additional options are described in the table below:

Option	Description
`protocol`	The maximum protocol version supported by the server. Currently this value is always 1.0.
`encoding`	The maximum encoding version supported by the server.

The encoding for UDP endpoints also changed in version 1.1.

Encoding for Endpoints

A proxy optionally contains an endpoint list or an adapter identifier, but not both:

- If a proxy contains endpoints, they are encoded immediately following the proxy options. A size specifying the number of endpoints is encoded first, followed by the endpoints. Each endpoint is encoded as a `short` specifying the endpoint type (`1`=TCP, `2`=SSL, `3`=UDP, 4=WS, 5=WSS), followed by an encapsulation of type-specific endpoint options. The type-specific options for TCP, UDP, SSL, WS and WSS are presented in the sections that follow.
- If a proxy does not have endpoints, a single byte with value `0` immediately follows the proxy options and a string representing the object adapter identifier is encoded immediately following the zero byte.
- For a proxy to a well-known object, which has neither endpoints nor an object adapter identifier, a single byte with value `0` immediat ely follows the proxy options and an empty string is encoded immediately following the zero byte.

Type-specific endpoint options are encapsulated because a receiver may not be capable of decoding them. For example, a receiver can only decode SSL endpoint options if it is configured with the IceSSL plug-in. However, the receiver must be able to re-encode the proxy with all of its original endpoints, in the order they were received, even if the receiver does not understand the type-specific options for an endpoint. Encapsulation of the endpoint into an opaque endpoint allows the receiver to do this.

Encoding for TCP Endpoints

A TCP endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct TCPEndpointData
{
    string host;
    int port;
    int timeout;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description
`host`	The server host (a host name or IP address)
`port`	The server port (1-65535)
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

Encoding for UDP Endpoints

The encoding format of UDP endpoints changed in version 1.1.

UDP Endpoint Encoding version 1.0

A UDP endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct UDPEndpointData
{
    string host;
    int port;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description
`host`	The server host (a host name or IP address)
`port`	The server port (1-65535)

`protocolMajor`	The major protocol version supported by the endpoint
`protocolMinor`	The highest minor protocol version supported by the endpoint
`encodingMajor`	The major encoding version supported by the endpoint
`encodingMinor`	The highest minor encoding version supported by the endpoint
`compress`	`true` if compression should be used (if possible), otherwise `false`

UDP Endpoint Encoding version 1.1

Version 1.1 of the encoding omits the protocol and encoding versions because these options are handled as proxy options instead:

Slice

```
struct UDPEndpointData
{
    string host;
    int port;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description
`host`	The server host (a host name or IP address)
`port`	The server port (1-65535)
`compress`	`true` if compression should be used (if possible), otherwise `false`

Encoding for SSL Endpoints

An SSL endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct SSLEndpointData
{
    string host;
    int port;
    int timeout;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description
`host`	The server host (a host name or IP address)

`port`	The server port (1-65535)
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

Encoding for WS Endpoints

A WebSocket endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct WSEndpointData
{
    string host;
    int port;
    int timeout;
    bool compress;
    string resource;
}
```

The endpoint options are described in the following table.

Option	Description
`host`	The server host (a host name or IP address)
`port`	The server port (1-65535)
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`
`resource`	A URI representing the web server resource associated with this endpoint

Encoding for WSS Endpoints

A secure WebSocket endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct WSSEndpointData
{
    string host;
    int port;
    int timeout;
    bool compress;
    string resource;
}
```

The endpoint options are described in the following table.

Option	Description

`host`	The server host (a host name or IP address)
`port`	The server port (1-65535)
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`
`resource`	A URI representing the web server resource associated with this endpoint

Encoding for BT Endpoints

A Bluetooth endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct BTEndpointData
{
    string addr;
    string uuid;
    int timeout;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description
`addr`	The Bluetooth address of the server
`uuid`	The UUID of the target service
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

Encoding for BTS Endpoints

A secure Bluetooth endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct BTSEndpointData
{
    string addr;
    string uuid;
    int timeout;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description

`addr`	The Bluetooth address of the server
`uuid`	The UUID of the target service
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

Encoding for iAP Endpoints

An iAP endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct IAPEndpointData
{
    string manufacturer;
    string modelNumber;
    string name;
    string protocol;
    int timeout;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description
`manufacturer`	The accessory manufacturer
`modelNumber`	The accessory model number
`name`	The accessory name
`protocol`	The protocol implemented by the accessory
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

Encoding for iAPS Endpoints

A secure iAP endpoint is encoded as an encapsulation containing the following structure:

Slice

```
struct IAPSEndpointData
{
    string manufacturer;
    string modelNumber;
    string name;
    string protocol;
    int timeout;
    bool compress;
}
```

The endpoint options are described in the following table.

Option	Description
`manufacturer`	The accessory manufacturer
`modelNumber`	The accessory model number
`name`	The accessory name
`protocol`	The protocol implemented by the accessory
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

See Also

- Object Identity
- Versioning
- Basic Data Encoding
- IceSSL
- Using Connections
- Protocol Compression

Data Encoding for Optional Values

Slice definitions can use optional values as parameters and as data members. To make optional values truly useful, the receiver needs the ability to gracefully ignore an optional value that it does not recognize. Although the Ice encoding omits type information and therefore is not self-describing, the encoding for optional values must include enough information for a receiver to determine how many bytes an optional value occupies so that it can skip to the next value in the stream. Despite this requirement, the encoding rules minimize the overhead associated with optional values, as you will see below.

On this page:

Overview of the Optional Value Encoding

The encoding for optional parameters and data members follows these general rules:

- Each optional value has a corresponding integer tag that uniquely identifies it.
- Optional values always appear *after* any required values.
- Optional values must be sorted by their tags. (Unlike required values, order of declaration does not affect optional values.)
- An optional value is encoded only if the sender has supplied a value.

Optional data members of a class or exception appear in the slice after any required data members and before the indirection table, if present. The slice flags must indicate the presence of optional data members, which are included in the byte count for the slice. If a slice contains optional data members, the byte value 255 must be written after the last optional data member; this marker denotes the end of the optional data members and is also included in the byte count for the slice.

The encoding does not use an explicit end marker for optional parameters; the end of the encapsulation also marks the end of any optional parameters.

An optional value is encoded as the tuple *<type, tag, value>*, where *type* is the *optional type* that tells the receiver how to determine the number of bytes occupied by the value. The value itself is marshaled using the standard Ice encoding rules for its Slice type.

> Optional values require encoding version 1.1.

Encoding for Optional Types and Tags

The first byte of an encoded optional value includes the optional type, and may also include the tag. The optional type occupies the first three bits of this byte, as described in the table below:

Name	Value	Description	Used for Slice type
F1	0	The value is encoded in one byte.	bool, byte
F2	1	The value is encoded in two bytes.	short
F4	2	The value is encoded in four bytes.	int, float
F8	3	The value is encoded in eight bytes.	double, long
Size	4	The value is encoded as a size.	enum
VSize	5	A leading size value indicates the number of bytes occupied by the value.	string, fixed-size structure, container of fixed-size elements
FSize	6	A leading 32-bit integer indicates the number of bytes occupied by the value.	variable-size structure, container of variable-size elements
Class	7	A class reference or inline instance.	class

The next five bits of the leading byte contain the tag, but only if the tag value is less than 30. Otherwise, the next five bits contain the value 30 as a marker to indicate that the tag value is encoded as a size starting with the next byte. As you can see, using tag values in the range 0 to 29 produces the most compact encoding.

Variable-size types whose encoded size cannot be determined in advance use the FSize optional type, where "FSize" denotes a leading fixed-length (32-bit) size. For these types, the sender reserves four bytes to hold the size, encodes the value as usual, then replaces the four bytes with the actual encoded size. This strategy avoids the need to shift the encoded data in the buffer, at the expense of potentially consuming more bytes than necessary to encode the size.

Fixed-size types use the VSize optional type, where "VSize" denotes a leading variable-length size. The sender can determine the encoded size of these types in advance, and therefore encodes it as a size followed by the value as usual.

Strings also use the VSize optional type but do not require an additional size because the string encoding already includes a leading size. The same is true for sequences of `bool` and `byte`.

The optional type Class represents a class reference or inline instance. Note that a receiver must decode an inline instance even if it does not recognize the tag value because the instance may be referenced by other parameters.

The following table describes the encoding of Slice types:

Slice type	Optional type	Data encoding	Notes
`bool`, `byte`	F1	value	
`short`	F2	value	
`int`, `float`	F4	value	
`long`, `double`	F8	value	
Proxy	FSize	int + data	32-bit integer holds the size of the encoded proxy
`class`	Class	reference or instance	
`enum`	Size	size	Enumerator encoded as a size
`string`	VSize	value	The encoded data for the string already contains a leading size
`sequence<bool>`, `sequence<byte>`	VSize	value	The encoded data for the sequence already contains a leading size
`sequence<`*fixed-size type*`>`, `dictionary<`*fixed-size key*`,`*fixed-size value*`>`	VSize	size + value	Size can be computed before encoding the container
`sequence<`*variable size type*`>`, `dictionary<`*variable-size key*`,`*variable-size value*`>`	FSize	int + value	32-bit integer holds the size of the container

Examples of Optional Value Encoding

The examples presented below demonstrate the encoding for optional values with typical use cases.

Optional Parameters Example

The following Slice operation makes use of optional input and output parameters:

Slice

```
bool op1(byte b,
         optional(2) string name,
         short sh,
         optional(1) long count,
         out double d,
         out optional(300) Object* p);
```

Suppose the parameters have the values shown in the table below:

Member	Type	Value	Marshaled size (in bytes)
b	byte	77	1
name	string	"joe"	4
sh	short	99	2
count	long	88	8
d	double	3.14	8
p	proxy	nil	2
return value	bool	true	1

The parameters of the outgoing request are encoded in an encapsulation with all required parameters first, in order of declaration, followed by the optional parameters sorted by tag:

Marshaled value	Size in bytes	Type	Byte offset
77 (*b*)	1	byte	0
99 (*sh*)	2	short	1
11 (*optional type F8 + tag 1*)	1	byte	3
88 (*count*)	8	long	4
21 (*optional type VSize + tag 2*)	1	byte	12
"joe" (*name*)	4	long	13

Using tag values less than 30 allows the encoding to combine the optional type and the tag into the same byte using the expression `(Tag << 3) + Type`.

Now consider the contents of the reply message:

Marshaled value	Size in bytes	Type	Byte offset
3.14 (*d*)	8	double	0
true (*return value*)	1	bool	8
246 (*optional type FSize + marker 30*)	1	byte	9
300 (*tag*)	5	size	10
2 (*32-bit FSize*)	4	int	15
nil (*p*)	2	Object*	19

The body of the reply message contains the out parameter d, the return value, and the optional parameter p. The tag value 300 is too large to combine with the optional type, therefore it appears immediately following the optional type encoded as a size. A proxy value uses the FSize optional type, meaning a 32-bit integer precedes the encoded value to specify its size.

> Although the return value is required in this example, an optional return value is treated as if it were an optional out parameter.

The server here supplies a nil value for the optional proxy parameter. The client must not interpret this to mean that the optional parameter is unset; rather, the parameter is set, it just happens to be set to a nil value. If the server had supplied no value for the parameter, it would not appear in the encoding at all.

As an example, if the client does not recognize the optional proxy parameter, it can skip the parameter as follows:

1. Extract the byte containing the optional type
2. Examine the first three bits to determine the type
3. Examine the five remaining bits to determine the tag's status
4. A value of 30 means the tag is encoded separately
5. Extract the tag as a size
6. The optional type FSize means a 32-bit integer holds the value's size; extract the integer
7. Skip ahead the specified number of bytes

8. If we have reached the end of the encapsulation, there are no more optional parameters remaining

Optional Data Members Example

The only difference between the encoding for optional parameters and the encoding for optional data members is the latter uses a marker byte to signify the end of optional parameters in a slice. Consider the following definitions:

Slice

```
struct Color
{
    short red;
    short green;
    short blue;
}

class Shape
{
    optional(1) string label;
}

class Rectangle extends Shape
{
    int width;
    int height;
    optional(10) Color fill;
    optional(9) Color border;
    optional(11) float scale;
}
```

Suppose we are encoding an instance of `Rectangle` with the member values shown in the table below:

Member	Type	Value	Marshaled size (in bytes)
`label`	`string`	`"r1"`	3
`width`	`int`	41	4
`height`	`int`	16	4
`fill`	`Color`	`0,0,0`	6
`border`	`Color`	`255,255,255`	6
`scale`	`float`	2.0	4

Using the sliced format, the instance data looks as follows:

Marshaled value	Size in bytes	Type	Byte offset
1 (*instance marker*)	1	`byte`	0
21 (*slice flags: string type ID, size is present, optional members are present*)	1	`byte`	1
`"::Rectangle"` (*type ID*)	12	`string`	2
34 (*byte count for slice*)	4	`int`	14
41 (*`width`*)	4	`int`	18

16 (*height*)	4	int	22
77 (*optional type VSize + tag 9*)	1	byte	26
6 (*VSize*)	1	size	27
{0,0,0} (*border*)	6	Color	28
85 (*optional type VSize + tag 10*)	1	byte	34
6 (*VSize*)	1	size	35
{255,255,255} (*fill*)	6	Color	36
90 (*optional type F4 + tag 11*)	1	byte	42
2.0 (*scale*)	4	float	43
255 (*end marker*)	1	byte	47
53 (*slice flags: string type ID, size is present, optional members are present, last slice*)	1	byte	48
"::Shape" (*type ID*)	8	string	49
9 (*byte count for slice*)	4	int	57
13 (*optional type VSize + tag 1*)	1	byte	61
"r1" (*label*)	3	string	62
255 (*end marker*)	1	byte	65

Notice the use of an end marker (byte value 255) denoting the end of the optional data members in each slice.

See Also

- Optional Values
- Basic Data Encoding
- Data Encoding for Classes

Protocol Messages

The Ice protocol uses five messages:

- Request (from client to server)
- Batch request (from client to server)
- Reply (from server to client)
- Validate connection (from server to client)
- Close connection (client to server or server to client)

Of these messages, validate and close connection only apply to connection-oriented transports.

As with the data encoding, protocol messages have no alignment restrictions. Each message consists of a message header and (except for validate and close connection) a message body that immediately follows the header.

On this page:

Message Header

Each protocol message has a 14-byte header that is encoded as if it were the following structure:

Slice

```
struct HeaderData
{
    int  magic;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    byte messageType;
    byte compressionStatus;
    int  messageSize;
}
```

The message header members are described in the following table.

Member	Description
`magic`	A four-byte magic number consisting of the ASCII-encoded values of 'I', 'c', 'e', 'P' (0x49, 0x63, 0x65, 0x50)
`protocolMajor`	The protocol major version number
`protocolMinor`	The protocol minor version number
`encodingMajor`	The encoding major version number
`encodingMinor`	The encoding minor version number
`messageType`	The message type

`compressionStatus`	The compression status of the message
`messageSize`	The size of the message in bytes, including the header

Currently, both the protocol and the encoding are at version 1.0. The valid message types are shown in the following table.

Message Type	Encoding
Request	`0`
Batch request	`1`
Reply	`2`
Validate connection	`3`
Close connection	`4`

The encoding for the message bodies of each of these message types is described in the sections that follow.

Request Message Body

A request message contains the data necessary to perform an invocation on an object, including the identity of the object, the operation name, and input parameters. A request message is encoded as if it were the following structure:

Slice

```
struct RequestData
{
    int requestId;
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
    byte mode;
    Ice::Context context;
    Encapsulation params;
}
```

The request members are described in the following table.

Member	Description
`requestId`	The request identifier
`id`	The object identity
`facet`	The facet name (zero- or one-element sequence)
`operation`	The operation name
`mode`	A byte representation of `Ice::OperationMode` (`0`=normal, `2`=idempotent)
`context`	The invocation context
`params`	The encapsulated input parameters, in order of declaration

The request identifier zero (`0`) is reserved for use in oneway requests and indicates that the server must not send a reply to the client. A non-zero request identifier must uniquely identify the request on a connection, and must not be reused while a reply for the identifier is outstanding.

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a request with a `facet` field with more than one element, it must throw a `MarshalException`.

Batch Request Message Body

A batch request message contains one or more oneway requests, bundled together for the sake of efficiency. A batch request message is encoded as integer (not a size) that specifies the number of requests in the batch, followed by the corresponding number of requests, encoded as if each request were the following structure:

Slice

```
struct BatchRequestData
{
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
    byte mode;
    Ice::Context context;
    Encapsulation params;
}
```

The batch request members are described in the following table.

Member	Description
`id`	The object identity
`facet`	The facet name (zero- or one-element sequence)
`operation`	The operation name
`mode`	A byte representation of `Ice::OperationMode`
`context`	The invocation context
`params`	The encapsulated input parameters, in order of declaration

Note that no request ID is necessary for batch requests because only oneway invocations can be batched.

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a batch request with a `facet` field with more than one element, it must throw a `MarshalException`.

Reply Message Body

A reply message body contains the results of a twoway invocation, including any return value, out-parameters, or exception. A reply message body is encoded as if it were the following structure:

Slice

```
struct ReplyData
{
    int requestId;
    byte replyStatus;
    Encapsulation body; // messageSize - 19 bytes
}
```

The first four bytes of a reply message body contain a request ID. The request ID matches an outgoing request and allows the requester to associate the reply with the original request.

The byte following the request ID indicates the status of the request; the remainder of the reply message body following the status byte is an encapsulation whose contents depend on the status value. The possible reply status values are shown in the table below (most of these values correspond to common exceptions).

Reply status	Success	Description
Success	`0`	A successful reply message is encoded as an encapsulation containing out-parameters (in the order of declaration), followed by the return value for the invocation, encoded according to their types as specified in Data Encoding. If an operation declares a `void` return type and no out-parameters, an empty encapsulation is encoded.
User exception	`1`	A user exception reply message contains an encapsulation containing the encoded user exception.
Object does not exist	`2`	If the target object does not exist, the reply message is encoded as if it were the following structure inside an encapsulation: **Slice** `struct ReplyData { Ice::Identity id; Ice::StringSeq facet; string operation; }` The invalid object reply members are described below: `id` — The object identity `facet` — The facet name (zero- or one-element sequence) `operation` — The operation name The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a reply with a `facet` field with more than one element, it must throw a `MarshalException`.
Facet does not exist	`3`	If the target object does not support the facet encoded in the request message, the reply message is encoded as for reply status 2.
Operation does not exist	`4`	If the target object does not support the operation encoded in the request message, the reply message is encoded as for reply status 2.
Unknown Ice local exception	`5`	The reply message for an unknown Ice local exception is encoded as an encapsulation containing a single string that describes the exception.

Unknown Ice user exception	6	The reply message for an unknown Ice user exception is encoded as an encapsulation containing a single string that describes the exception.
Unknown exception	7	The reply message for an unknown exception is encoded as an encapsulation containing a single string that describes the exception.

Validate Connection Message

A server sends a validate connection message when it receives a new connection.

> Validate connection messages are only used for connection-oriented transports.

The message indicates that the server is ready to receive requests; the client must not send any messages on the connection until it has received the validate connection message from the server. No reply to the message is expected by the server.

The purpose of the validate connection message is two-fold:

- It confirms to the client that the server is indeed an Ice server and not another type of server reached by accident.
- It prevents the client from writing a request message to its local transport buffers until after the server has acknowledged that it can actually process the request. This avoids a race condition caused by the server's TCP/IP stack accepting connections in its backlog while the server is in the process of shutting down: if the client were to send a request in this situation, the request would be lost but the client could not safely re-issue the request because that might violate at-most-once semantics. The validate connection message guarantees that a server is not in the middle of shutting down when the server's TCP/IP stack accepts an incoming connection and so avoids the race condition.

> As of Ice 3.6, validate connection messages may also be sent at any time by either side as a heartbeat. See Active Connection Management for more information.

The message header comprises the entire validate connection message. The compression status of a validate connection message is always `0`.

Close Connection Message

A close connection message is sent when a peer is about to gracefully shutdown a connection.

> Close connection messages are only used for connection-oriented transports.

The message header comprises the entire close connection message. The compression status of a close connection message is always `0`.

Either client or server can initiate connection closure. On the client side, connection closure is triggered by Active Connection Management (ACM), which automatically reclaims connections that have been idle for some time.

This means that connection closure can be initiated at will by either end of a connection; most importantly, no state is associated with a connection as far as the object model or application semantics are concerned.

The client side can close a connection whenever no reply for a request is outstanding on the connection. The sequence of events is:

1. The client sends a close connection message.
2. The client closes the writing end of the connection.
3. The server responds to the client's close connection message by closing the connection.

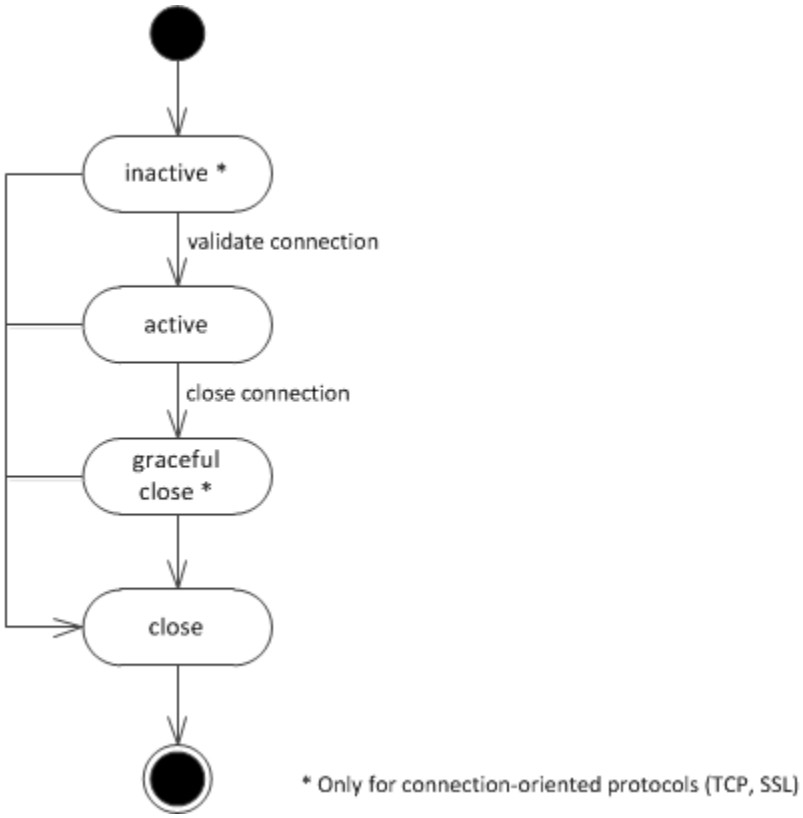
The server side can close a connection whenever no operation invocation is in progress that was invoked via that connection. This guarantees that the server will not violate at-most-once semantics: an operation, once invoked in a servant, is allowed to complete and its results are returned to the client. Note that the server can close a connection even after it has received a request from the client, provided that the request has not yet been passed to a servant. In other words, if the server decides that it wants to close a connection, the sequence of events is:

1. The server discards all incoming requests on the connection.
2. The server waits until all still executing requests have completed and their results have been returned to the client.
3. The server sends a close connection message to the client.

4. The server closes its writing end of the connection.
5. The client responds to the server's close connection message by closing both its reading and writing ends of the connection.
6. If the client has outstanding requests at the time it receives the close connection message, it re-issues these requests on a new connection. Doing so is guaranteed not to violate at-most-once semantics because the server guarantees not to close a connection while requests are still in progress on the server side.

Protocol State Machine

From a client's perspective, the Ice protocol behaves according to the state machine shown below:



Protocol state machine.

To summarize, a new connection is inactive until a validate connection message has been received by the client, at which point the active state is entered. The connection remains in the active state until it is shut down, which can occur when there are no more proxies using the connection, or after the connection has been idle for a while. At this point, the connection is gracefully closed, meaning that a close connection message is sent, and the connection is closed.

Disorderly Connection Closure

Any violation of the protocol or encoding rules results in a disorderly connection closure: the side of the connection that detects a violation unceremoniously closes it (without sending a close connection message or similar). There are many potential error conditions that can lead to disorderly connection closure; for example, the receiver might detect that a message has a bad magic number or incompatible version, receive a reply with an ID that does not match that of an outstanding request, receive a validate connection message when it should not, or find illegal data in a request (such as a negative size, or a size that disagrees with the actual data that was unmarshaled).

See Also

- Data Encoding
- Protocol Compression
- Object Identity
- Versioning
- Request Contexts
- Oneway Invocations

Protocol Compression

On this page:

Overview of Protocol Compression

Compression is an optional feature of the Ice protocol; whether it is used for a particular message is determined by several factors:

1. Compression may not be supported on all platforms or in all language mappings.
2. Compression can be used in a request or batch request only if the peer advertises the ability to accept compressed messages.
3. For efficiency reasons, the Ice protocol engine does not compress messages smaller than 100 bytes.

> A compliant implementation of the protocol is free to compress messages that are smaller than 100 bytes — the choice is up to the protocol implementation.

> Compression is likely to improve performance only over lower-speed links, for which bandwidth is the overall limiting factor. Over high-speed LAN links, the CPU time spent on compressing and uncompressing messages is longer than the time it takes to just send the uncompressed data.

Encoding for Compressed Messages

If compression is used, the entire protocol message excluding the header is compressed using the bzip2 algorithm [1]. The `messageSize` member of the message header therefore reflects the size of the compressed message, including the uncompressed header, plus an additional four bytes.

The `compressionStatus` field of the message header indicates whether a message is compressed and provides additional information, as shown in the table below.

Compression status	Encoding	Applies to	Description
Message is uncompressed. Client cannot accept compressed replies.	`0`	Request, Batch Request, Reply, Validate Connection, Close Connection	A client that does not support compression always uses this value. A client that supports compression sets the value to `0` if the endpoint via which the request is dispatched indicates that it does not support compression. A server uses this value for uncompressed replies.
Message is uncompressed. Client can accept compressed replies.	`1`	Request, Batch Request	A client uses this value if the endpoint via which the request is dispatched indicates that it supports compression, but the client has decided not to use compression for this particular request (presumably because the request is too small, so compression does not provide any saving).
Message is compressed.	`2`	Request, Batch Request, Reply	A client that supports compression sets this value only if the endpoint via which the request is dispatched indicates that it supports compression. A server uses this value for compressed replies.

The message body of a compressed request, batch request, or reply message is encoded by first writing the size of the uncompressed message (including its header) as a four-byte integer, followed by the compressed message body (excluding the header). It follows that the size of a compressed message is 14 bytes for the header, plus four bytes to record the size of the uncompressed message, plus the number of bytes occupied by the compressed message body. Writing the uncompressed message size prior to the body enables the receiver to allocate a buffer that is large enough to accomodate the uncompressed message body.

Compression Semantics for Clients

A client sends a compressed message if the following are true:

- The client-side run time supports compression
- The size of the uncompressed message is at least 100 bytes

- The proxy endpoint on which the message will be sent has the compression flag (`-z` for stringified endpoints) indicating that the server supports compression

If any of these criteria are false, the client sends an uncompressed message. In either case, the client uses the `compressionStatus` field of the message header to inform the server whether the client supports compression.

Compression Semantics for Servers

A server will only receive a compressed message if the client's proxy endpoint has the compression flag (`-z` for stringified endpoints). For simpler situations where a client is constructing proxies itself, such as obtaining them from a configuration file, you only need to ensure that the proxy's endpoints enable the compression flag. On the other hand, if the server is dynamically constructing proxies and returning them to the client for eventual invocations, then the server should also enable the compression flag in its object adapter endpoints as shown in the example configuration below:

```
MyAdapter.Endpoints=tcp -h 192.168.1.17 -p 2500 -z
```

Specifying the flag here causes every proxy created by the object adapter to advertise compression-capable endpoints.

A server examines the `compressionStatus` field of an incoming message header not only to determine whether the message itself is compressed but also to discover whether the client supports compressed replies. A server sends a compressed reply if the following are true:

- The server-side run time supports compression
- The size of the uncompressed reply is at least 100 bytes
- The `compressionStatus` field of the corresponding request message has a value of 1 or 2

If any of these criteria are false, the server sends an uncompressed reply.

See Also

- Data Encoding for Proxies
- Protocol Messages

References

1. Red Hat, Inc. 2003. *The bzip2 and libbzip2 Home Page*. Raleigh, NC: Red Hat, Inc.

Protocol and Encoding Versions

On this page:

Version Flexibility

As we saw in the preceding sections, both the Ice protocol and encoding have separate major and minor version numbers. Separate versioning of protocol and encoding has the advantage that neither depends on the other: any version of the Ice protocol can be used with any version of the encoding, so they can evolve independently. For example, you could send a request with Ice protocol version 1.1, and encode the parameters with encoding version 2.3.

The Ice versioning mechanism provides the maximum possible amount of interoperability between clients and servers that use different versions of the Ice run time. In particular, older deployed clients can communicate with newer deployed servers and vice versa, provided that the message contents use types that are understandable to both sides. Intermediary servers, such as a Glacier2 router, do not and should not check the contents of the requests and responses they forward; this way, an older intermediary server can forward messages encoded with a newer encoding that it does not understand.

> Due to a bug, a Glacier2 router prior to version 3.5 can only forward messages encoded using the 1.0 encoding.

Ice 3.5 introduced a new encoding, encoding version 1.1, in support of several new features such as the ability to define optional parameters in operations. In order to send a request with an optional parameter, you need to use encoding 1.1 or greater.

Ice 3.5 (or newer) is capable of marshaling and unmarshaling messages with both the 1.0 and 1.1 encodings (it uses the 1.1 encoding by default). This allows interoperability to the maximum extent possible:

Client Ice Version	Server Ice Version	Encoding Version	Operation without Optional Parameter	Operation with Optional Parameter
3.4 or earlier	3.4 or earlier	1.0	Yes	No
3.4 or earlier	3.5 or newer	1.0	Yes	No
3.5 or newer	3.4 or earlier	1.0	Yes	No
3.5 or newer	3.5 or newer	1.1	Yes	Yes

When an Ice 3.5 (or newer) client sends a request to an Ice 3.4 server, it must naturally use the 1.0 encoding. This is achieved by setting correctly (and often automatically) the encoding version on the proxy used. If this proxy is received by the client through a response using the 1.0 encoding, the decoded proxy will automatically have the desired 1.0 encoding (all proxies in a 1.0-encoded message implicitly have 1.0 as their encoding version and 1.0 as their protocol version). If this proxy is created with `stringToProxy` or `propertyToProxy`, the client will need to set this proxy's encoding version to 1.0 through one of the following methods:

- with `-e 1.0` in the source stringified proxy
- by creating a new proxy with the desired encoding version, by calling `ice_encodingVersion("1.0")` on the proxy
- by setting `Ice.Default.EncodingVersion` to 1.0.

Version Ground Rules

For versioning of the protocol and encoding to be possible, all versions (present and future) of the Ice run time adhere to a few ground rules:

1. Encapsulations always have a six-byte header; the first four bytes are the size of the encapsulation (including the size of the header), followed by two bytes that indicate the major and minor version. How to interpret the remainder of the encapsulation depends on the major and minor version.
2. The first eight bytes of a message header always contain the magic number 'I', 'c', 'e', 'P', followed by four bytes of version information (two bytes for the protocol major and minor number, and two bytes for the encoding major and minor number). How to interpret the remainder of the header and the message body depends on the major and minor version.

These ground rules ensure that all current and future versions of the Ice run time can at least identify the version and size of an encapsulation and a message. This is particularly important for message switches such as IceStorm; by keeping the version and size information in a fixed format, it is possible to forward messages that are, for example, at version 2.0, even though the message switch itself may still be at version 1.0.

See Also

- Basic Data Encoding
- Protocol Messages
- IceStorm
- IceGrid

Best Practices

This section provides a number of design guidelines for Ice applications.

Topics

Optional Values

Version 1.1 of the Ice encoding introduces a convenient new feature that allows you to declare data members and parameters as optional. Collectively called *optional values*, this feature provides two main benefits for Ice applications:

- It offers a simple and natively-supported mechanism for representing values that may or may not be present, without consuming any additional space in a message when an optional value is not used.
- It significantly improves the ability to gradually evolve components of an Ice application while maintaining backward compatibility with existing deployments.

On this page:

Overview of Optional Values

The optional syntax is the same for both parameters and data members:

`optional(`*`tag`*`)` *`type name`*

The `optional` keyword denotes an optional value with the given tag. The value for *`tag`* must be a non-negative integer.

> If your Slice definitions currently use `optional` as an identifier, you can continue using it as an identifier by escaping it as `\optional`.

Data members and parameters not marked as optional are considered *required*; a sender must supply values for all required parameters and members.

Each language mapping defines APIs for passing an optional value, indicating that an optional value is not set, and testing whether an optional value is set. For example, the three strongly-typed languages that Ice currently supports (C++, C#, and Java) all use a similar API: an `Optional` type wraps the optional value and provides methods for examining and changing its value. Refer to the language mapping sections for more details on optional values.

> A well-behaved program must not assume that an optional data member or parameter always has a value.

An optional value incurs no additional overhead in the encoding when its value is unset. If an optional parameter or member has a value, its overhead depends on several factors, including your choice of tag values.

Optional Parameters in Operations

An operation may declare as optional its return value and any of its parameters. The ordering rules for parameters remain unchanged: all input parameters must precede the output parameters.

The following Slice operation declares a mix of optional and required parameters, as well as an optional return value:

Slice

```
interface Database
{
    optional(3) Record
    lookup(string key, optional(1) Filter f, out optional(2) Cursor
matches);
}
```

We can invoke `lookup` in C++ as shown below:

C++11

```
std::shared_ptr<DatabasePrx> proxy = ...;
Ice::optional<Cursor> matches;
auto rec = proxy->lookup("someKey", Ice::nullopt, matches);
if(rec)
{
    // lookup returned a value

    if(matches)
    {
        // there are more matches
    }
}
```

Optional Data Members in Classes and Exceptions

Data members can be declared as optional in classes and exceptions. Structures do not support optional data members.

Consider the following example:

Slice

```
class Account
{
    string accountNo;
    optional(9) Account referrer;
}
```

We can use `Account` in C# as shown below:

C#

```
void printAccount(Account acct)
{
    Console.WriteLine("Account #: " + acct.accountNo);
    if(acct.referrer.HasValue)
    {
        Console.WriteLine("Referrer: " + acct.referrer.Value.accountNo);
    }
    else
    {
        Console.WriteLine("Referrer: None");
    }
}

Account a1 = new Account("100-21807", Ice.Util.None);
printAccount(a1);
Account a2 = new Account("165-75122", a1);
printAccount(a2);
```

Implicit conversion operators in C# allow us to pass a value of the declared type, or `Ice.Util.None` for an unset value, where an optional value is expected.

Guidelines for using Optional Values

Keep the following guidelines in mind while using optional values:

- Consider an optional value as a logical unit composed of its tag and type. The parameter or member name is not included in the Ice encoding, therefore changes to the name do not affect on-the-wire compatibility. Changing the tag of a value is equivalent to adding a new value: the previous iteration of the value still exists as long as there is at least one deployed program that might use it.
- To change the type of an optional value without affecting compatibility with existing deployments, you must also change its tag. You are essentially deprecating the old value and adding a new one.
- Whenever you change the tag of a value, we recommend making a note of the old definition (such as in a Slice comment) so that you do not accidentally reuse its tag.
- Adding a new optional parameter or member does not affect compatibility with existing deployments as long as its tag is not currently in use.
- Removing an optional parameter or member does not affect compatibility with existing deployments if programs are written correctly to test for the presence of the value prior to using it.
- The order of declaration is important for required members and parameters, but is not important for optional members and parameters. If you wish to maintain compatibility with existing deployments, do not change the order of declaration for required members and parameters. Compatibility is not affected by changes to the order of optional members and parameters.

Backward Compatibility with Encoding Version 1.0

Optional values require version 1.1 of the Ice encoding. If the intended receiver only supports version 1.0 of the encoding, the Ice run time in the sender will omit all optional parameters when marshaling the operation's parameters or results, and omit all optional data members. This behavior makes it possible for you to add optional parameters to an operation, or optional members to a class or exception, without affecting backward compatibility with existing deployments that use version 1.0 of the encoding, while gaining the ability to exchange optional values with new deployments that use version 1.1 of the encoding.

For example, suppose existing deployments use the following class definition:

Slice

```
// Version 1
class Person
{
    string firstName;
    string lastName;
}
```

We can safely add an optional member without affecting compatibility:

Slice

```
// Version 2
class Person
{
    string firstName;
    string lastName;
    optional(1) Date birthDate;
}
```

It is even safe to insert an optional member between two existing members:

Slice

```
// Version 3
class Person
{
    string firstName;
    optional(2) string middleName;
    string lastName;
    optional(1) Date birthDate;
}
```

When sending an instance of `Person` to a receiver that supports only encoding version 1.0, the optional members are omitted and the encoded form matches our initial version of the class.

A similar situation exists for operations. For example, the following three operations are compatible when using encoding version 1.0:

Slice

```
Person createPerson(string firstName, string lastName); // V1

Person createPerson(string firstName, string lastName, optional(1) Date
birthDate); // V2

Person createPerson(string firstName, optional(2) string middleName,
string lastName,
                    optional(1) Date birthDate); // V3
```

Note however that other types of changes can easily break compatibility.

Using Optional Values Efficiently

The Ice encoding uses a self-describing format to minimize the overhead associated with optional data members and allow for efficient decoding in a receiver. The amount of overhead required for a member depends on the member's type and its tag. The type contributes between one and five bytes of overhead, while the tag value contributes an additional zero to five bytes. Tag values less than 30 add no additional overhead; tag values between 30 and 254 add a single byte, and tag values of 255 or greater add five bytes.

> The overhead associated with the member's type is out of your control, but you have direct control of the overhead for tags. To minimize this overhead, use tags in the range zero to 29.

The table below describes the overhead associated with several common Slice types:

Type	Overhead (bytes)
Primitive (including `string`)	1
Proxy	5
Object	1
Sequence of `byte`, `bool`	1
Sequence	2-5
Enumerator	1
Structure	2-5
Dictionary	2-5

Refer to the encoding specification for more details on marshaling optional data members.

See Also

- Data Encoding
- Optional Data Members
- Optional Parameters and Return Values

Server Implementation Techniques

As mentioned in Servant Locators, instantiating a servant for each Ice object on server start-up is a viable design, provided that you can afford the amount of memory required by the servants, as well as the delay in start-up of the server. However, Ice supports more flexible mappings between Ice objects and servants; these alternate mappings allow you to precisely control the trade-off between memory consumption, scalability, and performance. We outline a few of the more common implementation techniques here.

On this page:

Incremental Server Initialization

If you use a servant locator, the servant returned by `locate` is used only for the current request, that is, the Ice run time does not add the servant to the Active Servant Map (ASM). Of course, this means that if another request comes in for the same Ice object, `locate` must again retrieve the object state and instantiate a servant. A common implementation technique is to add each servant to the ASM as part of `locate`. This means that only the first request for each Ice object triggers a call to `locate`; thereafter, the servant for the corresponding Ice object can be found in the ASM and the Ice run time can immediately dispatch another incoming request for the same Ice object without having to call the servant locator.

An implementation of `locate` to do this would look something like the following:

C++11

```
std::shared_ptr<Ice::Object>
MyServantLocator::locate(const Ice::Current& current,
std::shared_ptr<void>& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = current.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try
    {
        d = DB_lookup(name);
    }
    catch(const DB_error&)
    {
        return 0;
    }

    // We have the state, instantiate a servant.
    //
    auto servant = std::make_shared<PhoneEntryI>(d);

    // Add the servant to the ASM.
    //
    current.adapter->add(servant, current.id);      // NOTE: incorrect!

    return servant;
}
```

This is almost identical to the implementation seen in our earlier example — the only difference is that we also add the servant to the ASM by calling `ObjectAdapter::add`. Unfortunately, this implementation is wrong because it suffers from a race condition. Consider the situation where we do not have a servant for a particular Ice object in the ASM, and two clients more or less simultaneously send a request for the same Ice object. It is entirely possible for the thread scheduler to schedule the two incoming requests such that the Ice run time completes the lookup in the ASM for both requests and, for each request, concludes that no servant is in memory. The net effect is that `locate` will be called twice for the same Ice object, and our servant locator will instantiate two servants instead of a single servant. Because the second call to `ObjectAdapter::add` will raise an `AlreadyRegisteredException`, only one of the two servants will be added to the ASM.

Of course, this is hardly the behavior we expect. To avoid the race condition, our implementation of `locate` must check whether a concurrent invocation has already instantiated a servant for the incoming request and, if so, return that servant instead of instantiating a new one. The Ice run time provides the `ObjectAdapter::find` operation to allow us to test whether an entry for a specific identity already exists in the ASM:

Slice

```
module Ice
{
    local interface ObjectAdapter
    {
        // ...
        Object find(Identity id);
        // ...
    }
}
```

`find` returns the servant if it exists in the ASM and null, otherwise. Using this lookup function, together with a mutex, allows us to correctly implement `locate`. The class definition of our servant locator now has a private mutex so we can establish a critical region inside `locate`:

C++11

```
class MyServantLocator : public Ice::ServantLocator
{
public:

    virtual std::shared_ptr<Ice::Object> locate(const Ice::Current& c,
std::shared_ptr<void>&) override;

    // Declaration of finished() and deactivate() here...

private:
    std::mutex _m;
};
```

The `locate` member locks the mutex and tests whether a servant is already in the ASM: if so, it returns that servant; otherwise, it instantiates a new servant and adds it to the ASM as before:

C++11

```
std::shared_ptr<Ice::Object>
MyServantLocator::locate(const Ice::Current& current,
std::shared_ptr<void>& cookie)
{
    std::lock_guard<std::mutex> lock(_m);

    // Check if we have instantiated a servant already.
    //
    auto servant = current.adapter.find(current.id);

    if(!servant)     // We don't have a servant already
    {
        // Instantiate a servant.
        //
        ServantDetails d;
        try
        {
           d = DB_lookup(current.id.name);
        }
        catch(const DB_error&)
        {
           return 0;
        }
        servant = std::make_shared<PhoneEntryI>(d);

        // Add the servant to the ASM.
        //
        current.adapter->add(servant, current.id);
    }

    return servant;
}
```

The Java version of this locator is almost identical, but we use the `synchronized` qualifier instead of a mutex to make `locate` a critical region:

Java

```
import com.zeroc.Ice.ServantLocator;

...

synchronized public ServantLocator.LocateResult
locate(com.zeroc.Ice.Current c)
{
    ServantLocator.LocateResult r = new ServantLocator.LocateResult();
    // Check if we have instantiated a servant already.
    //
    com.zeroc.Ice.Object servant = c.adapter.find(c.id);

    if(servant == null) // We don't have a servant already
    {
        // Instantiate a servant
        //
        ServantDetails d;
        try
        {
            d = DB.lookup(c.id.name);
        }
        catch(DB.error e)
        {
            return r;
        }
        servant = new PhoneEntryI(d);

        // Add the servant to the ASM.
        //
        c.adapter.add(servant, c.id);
     }

    r.returnValue = servant;
    return r;
}
```

In C#, you can place the body of `locate` into a `lock(this)` statement.

Using a servant locator that adds the servant to the ASM has a number of advantages:

- Servants are instantiated on demand, so the cost of initializing the servants is spread out over many invocations instead of being incurred all at once during server start-up.
- The memory requirements for the server are reduced because servants are instantiated only for those Ice objects that are actually accessed by clients. If clients only access a subset of the total number of Ice objects, the memory savings can be substantial.

In general, incremental initialization is beneficial if instantiating servants during start-up is too slow. The memory savings can be worthwhile as well but, as a rule, are realized only for comparatively short-lived servers: for long-running servers, chances are that, sooner or later, every Ice object will be accessed by some client or another; in that case, there are no memory savings because we end up with an instantiated servant for every Ice object regardless.

Implementing a Server using Default Servants

Default servants are a very effective tool for conserving memory when a server hosts a large number of Ice objects.

To create a default servant implementation, we need as many default servants as there are non-abstract interfaces in the system. For example, for our file system application, we require two default servants, one for directories and one for files. In addition, the object identities we create use the `category` member of the object identity to encode the type of interface of the corresponding Ice object. The value of the category field can be anything that identifies the interface, such as the 'd' and 'f' convention we suggested earlier. Alternatively, you could use `"Directory"` and `"File"`, or use the type ID of the corresponding interface, such as `"::Filesystem::Directory"` and `"::File system::File"`. The `name` member of the object identity must be set to whatever identifier we can use to retrieve the persistent state of each directory and file from secondary storage. (For our file system application, we used a UUID as a unique identifier.)

Registration of the default servants is as follows:

C++11

```
adapter->addDefaultServant(make_shared<DirectoryI>(), "d");
adapter->addDefaultServant(make_shared<FileI>(), "f");
```

All the action happens in the implementation of the operations, using the following steps for each operation:

1. Use the passed `Current` object to get the identity for the current request.
2. Use the `name` member of the identity to locate the persistent state of the servant on secondary storage. If no record can be found for the identity, throw an `ObjectNotExistException`.
3. Implement the operation to operate on that retrieved state (returning the state or updating the state as appropriate for the operation).

This might look something like the following:

C++11

```
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current& current) const override
{
    // Use the identity of the directory to retrieve
    // its contents.
    DirectoryContents dc;
    try
    {
        dc = DB_getDirectory(current.id.name);
    }
    catch(const DB_error&)
    {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }

    // Use the records retrieved from the database to
    // initialize return value.
    //
    FileSystem::NodeSeq ns;
    // ...

    return ns;
}
```

Note that the servant implementation is completely stateless: the only state it operates on is the identity of the Ice object for the current request (and that identity is passed as part of the `Current` parameter).

Overriding `ice_ping`

We recommended that a default servant implementation take steps to preserve the semantics of the `ice_ping` operation, which is used to test whether an Ice object exists. If a default servant fails to override `ice_ping`, clients may mistakenly believe that a non-existent Ice object still exists. The code below demonstrates how we can override the operation in our file system application:

C++11

```
void
Filesystem::DirectoryI::ice_ping(const Ice::Current& current) const
override
{
    try
    {
        d = DB_lookup(current.id.name);
    }
    catch(const DB_error&)
    {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

It is good practice to override `ice_ping` if you are using default servants.

Combining Server Implementation Techniques

Depending on the nature of your application, you may be able to steer a middle path that provides better performance while keeping memory requirements low: if your application has a number of frequently-accessed objects that are performance-critical, you can add servants for those objects to the ASM. If you store the state of these objects in data members inside the servants, you effectively have a cache of these objects.

The remaining, less-frequently accessed objects can be implemented with a default servant. For example, in our file system implementation, we could choose to instantiate directory servants permanently, but to have file objects implemented with a default servant. This provides efficient navigation through the directory tree and incurs slower performance only for the (presumably less frequent) file accesses.

This technique could be augmented with a cache of recently-accessed files, along similar lines to the buffer pool used by the Unix kernel [1]. The point is that you can combine use of the ASM with servant locators and default servants to precisely control the trade-offs among scalability, memory consumption, and performance to suit the needs of your application.

See Also

- Servant Locators
- Servant Locator Example
- Servant Activation and Deactivation
- Using Identity Categories with Servant Locators
- Default Servants

References

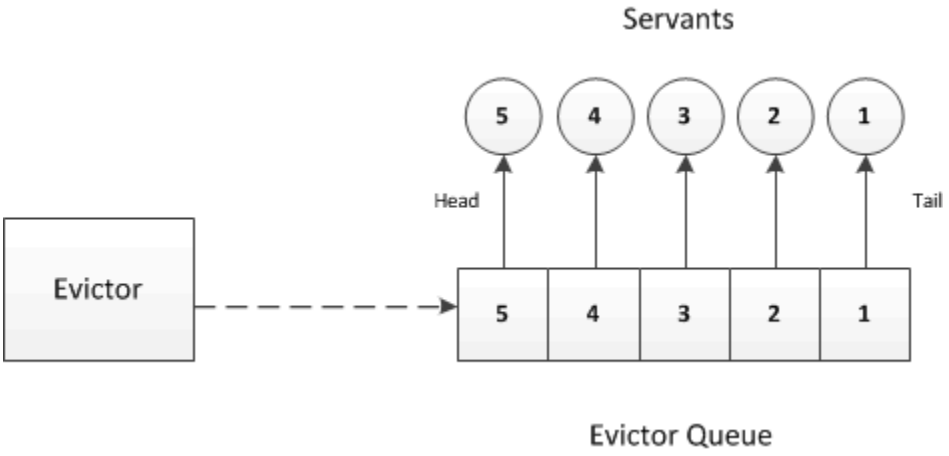
1. McKusick, M. K., et al. 1996. The Design and Implementation of the 4.4BSD Operating System. Reading, MA: Addison-Wesley.

Servant Evictors

A particularly interesting use of a servant locator is as an *evictor* [1]. An evictor is a servant locator that maintains a cache of servants:

- Whenever a request arrives (that is, `locate` is called by the Ice run time), the evictor checks to see whether it can find a servant for the request in its cache. If so, it returns the servant that is already instantiated in the cache; otherwise, it instantiates a servant and adds it to the cache.
- The cache is a queue that is maintained in least-recently used (LRU) order: the least-recently used servant is at the tail of the queue, and the most-recently used servant is at the head of the queue. Whenever a servant is returned from or added to the cache, it is moved from its current queue position to the head of the queue, that is, the "newest" servant is always at the head, and the "oldest" servant is always at the tail.
- The queue has a configurable length that corresponds to how many servants will be held in the cache; if a request arrives for an Ice object that does not have a servant in memory and the cache is full, the evictor removes the least-recently used servant at the tail of the queue from the cache in order to make room for the servant about to be instantiated at the head of the queue.

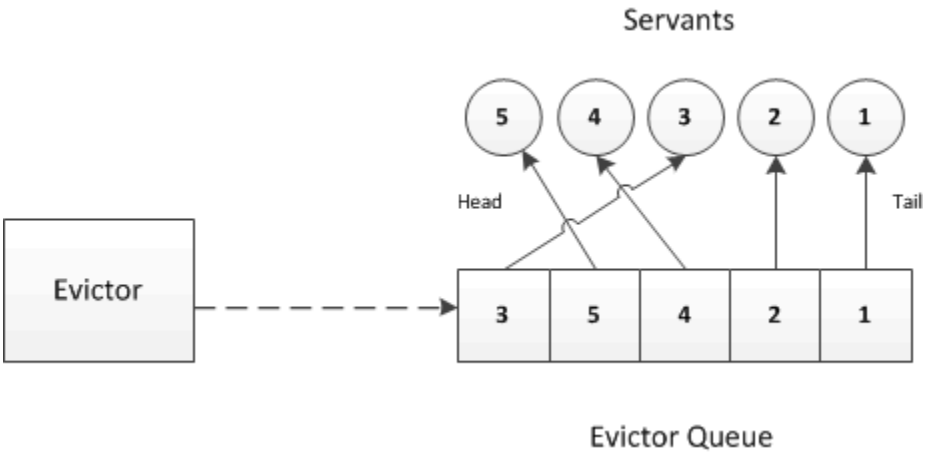
The figure below illustrates an evictor with a cache size of five after five invocations have been made, for object identities 1 to 5, in that order.



An evictor after five invocations for object identities 1 to 5.

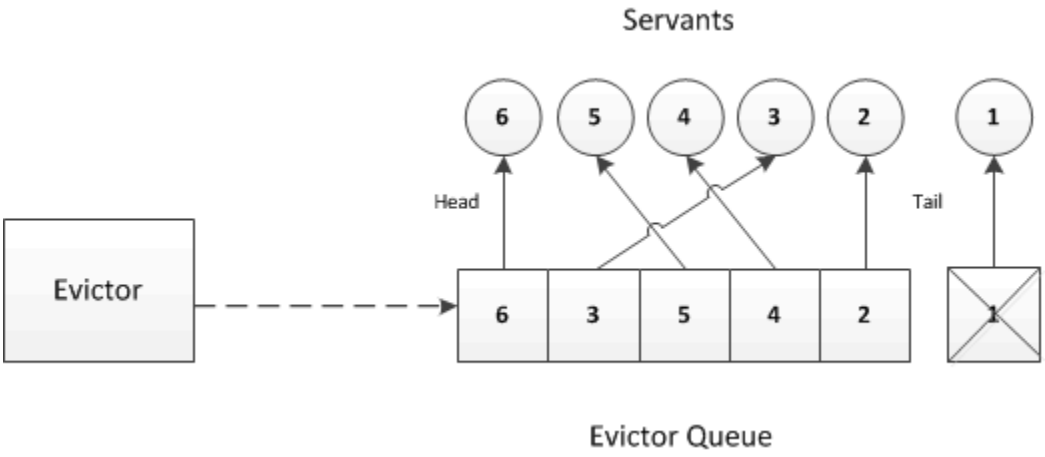
At this point, the evictor has instantiated five servants, and has placed each servant onto the evictor queue. Because requests were sent by the client for object identities 1 to 5 (in that order), servant 5 ends up at the head of the queue (at the most-recently used position), and servant 1 ends up at the tail of the queue (at the least-recently used position).

Assume that the client now sends a request for servant 3. In this case, the servant is found on the evictor queue and moved to the head position. The resulting ordering is shown below:



The evictor after accessing servant 3.

Assume that the next client request is for object identity 6. The evictor queue is fully populated, so the evictor creates a servant for object identity 6, places that servant at the head of the queue, and evicts the servant with identity 1 (the least-recently used servant) at the tail of the queue, as you can see here:



The evictor after evicting servant 1.

The evictor pattern combines the advantages of the ASM with the advantages of a default servant: provided that the cache size is sufficient to hold the working set of servants in memory, most requests are served by an already instantiated servant, without incurring the overhead of creating a servant and accessing the database to initialize servant state. By setting the cache size, you can control the trade-off between performance and memory consumption as appropriate for your application.

The following pages show how to implement an evictor in several languages. (You can also find the source code for the evictor with the code examples for this manual in the Ice distribution.)

Topics

- Implementing a Servant Evictor in C++
- Implementing a Servant Evictor in Java
- Implementing a Servant Evictor in C-Sharp

See Also

- Servant Locators
- Default Servants

References

1. Henning, M., and S. Vinoski. 1999. Advanced CORBA Programming with C++. Reading, MA: Addison-Wesley.

Implementing a Servant Evictor in C++

On this page:

The `EvictorBase` Class in C++

The evictor we show here is designed as an abstract base class: in order to use it, you derive a class from the `EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definition as follows:

C++11

```
class EvictorBase : public Ice::ServantLocator
{
public:

    EvictorBase(int size = 1000);

    virtual std::shared_ptr<Ice::Object> locate(const Ice::Current&,
std::shared_ptr<void>&) override;
    virtual void finished(const Ice::Current&, const
std::shared_ptr<Ice::Object>&, const std::shared_ptr<void>&) override;
    virtual void deactivate(const std::string&) override;

protected:

    virtual std::shared_ptr<Ice::Object> add(const Ice::Current&,
std::shared_ptr<void>&) = 0;
    virtual void evict(const std::shared_ptr<Ice::Object>&, const
std::shared_ptr<void>&) = 0;


...
};
```

Note that the evictor has a constructor that sets the size of the queue, with a default argument to set the size to 1000.

The `locate`, `finished`, and `deactivate` functions are inherited from the `ServantLocator` base class; these functions implement the logic to maintain the queue in LRU order and to add and evict servants as needed.

The `add` and `evict` functions are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are pure virtual, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant, allowing the subclass to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map not only stores servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.

2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue. Storing the queue position is not strictly necessary — we store the position for efficiency reasons because it allows us to locate a servant's position in the queue in constant time instead of having to search through the queue in order to maintain its LRU property.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The need for the use count deserves some extra explanation: suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

The evictor queue does not store the identity of the servant. Instead, the entries on the queue are iterators into the evictor map. This is useful when the time comes to evict a servant: instead of having to search the map for the identity of the servant to be evicted, we can simply delete the map entry that is pointed at by the iterator at the tail of the queue. We can get away with storing an iterator into the evictor queue as part of the map, and storing an iterator into the evictor map as part of the queue because both `std::list` and `std::map` do not invalidate forward iterators when we add or delete entries (except for invalidating iterators that point at a deleted entry, of course).

> Reverse iterators *can* be invalidated by modification of list entries: if a reverse iterator points at `rend` and the element at the head of the list is erased, the iterator pointing at `rend` is invalidated.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

C++11

```
class EvictorBase : public Ice::ServantLocator
{
    // ...

private:

    struct EvictorEntry;

    using EvictorMap = std::map<Ice::Identity,
std::shared_ptr<EvictorEntry>>;
    using EvictorQueue = std::list<EvictorMap::iterator>;

    struct EvictorEntry
    {
        std::shared_ptr<Ice::Object> servant;
        std::shared_ptr<void> userCookie;
        EvictorQueue::iterator queuePos;
        int useCount;
    };

    EvictorMap _map;
    EvictorQueue _queue;
    int _size;

    std::mutex _mutex;

    void evictServants();
};
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. In addition, we use a private `_mutex` data member so we can correctly serialize access to the evictor's data structures.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit — we will discuss this function in more detail shortly.

The `EvictorEntry` structure serves as the cookie that we pass from `locate` to `finished`; it stores the servant, the servant's position in the evictor queue, the servant's use count, and the cookie that we pass from `add` to `evict`.

The implementation of the constructor is trivial. The only point of note is that we ignore negative sizes:

C++11

```
EvictorBase::EvictorBase(int size)
   : _size(size)
{
    if(_size < 0)
    {
        _size = 1000;
    }
}
```

We could have stored the size as a `size_t` instead. However, for consistency with the Java implementation, which cannot use unsigned integers, we use `int` to store the size.

Almost all the action of the evictor takes place in the implementation of `locate`:

C++11

```
shared_ptr<Ice::Object>
EvictorBase::locate(const Ice::Current& c, shared_ptr<void>& cookie)
{
    lock_guard<mutex> lk(_mutex);

    //
    // Check if we have a servant in the map already.
    //
    shared_ptr<EvictorEntry> entry;
    auto i = _map.find(c.id);
    if(i != _map.end())
    {
        //
        // Got an entry already, dequeue the entry from its current
position.
        //
        entry = i->second;
        _queue.erase(entry->queuePos);
    }
    else
    {
        //
        // We do not have an entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        entry = make_shared<EvictorEntry>();
        entry->servant = add(c, entry->userCookie); // Down-call
        if(!entry->servant)
        {
            return 0;
        }
        entry->useCount = 0;
        i = _map.insert(make_pair(c.id, entry)).first;
    }

    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(entry->useCount);
    entry->queuePos = _queue.insert(_queue.begin(), i);

    cookie = entry;

    return entry->servant;
}
```

The first step in `locate` is to lock the `_mutex` data member. This protects the evictor's data structures from concurrent access. The next step is to instantiate a smart pointer to an `EvictorEntry`. That smart pointer acts as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. That same smart pointer is also the value type of our map entries, so we do not store two copies of the same information redundantly — instead, smart pointers ensure that a single copy of each `EvictorEntry` structure is shared by both the cookie and the map.

The next step is to look in the evictor map to see whether we already have an entry for this object identity. If so, we remove the entry from its current queue position.

Otherwise, we do not have an entry for this object identity yet, so we have to create one. The code creates a new evictor entry, and then calls `add` to get a new servant. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we return zero to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The last few lines of `locate` add the entry for the current request to the head of the evictor queue to maintain its LRU property, increment the use count of the entry, set the cookie that is returned from locate to point at the `EvictorEntry`, and finally return the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

C++11

```
void
EvictorBase::finished(const Ice::Current&, const
shared_ptr<Ice::Object>&, const shared_ptr<void>& cookie)
{
    lock_guard<mutex> lk(_mutex);

    auto entry = static_pointer_cast<EvictorEntry>(cookie);

    //
    // Decrement use count and check if there is something to evict.
    //
    --(entry->useCount);
    evictServants();
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

C++11

```
void
EvictorBase::evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    auto p = _queue.rbegin();
    auto excessEntries = static_cast<int>(_map.size() - _size);

    for(int i = 0; i < excessEntries; ++i)
    {
        auto mapPos = *p;
        if(mapPos->second->useCount == 0)
        {
            evict(mapPos->second->servant, mapPos->second->userCookie);
// Down-call
            p =
EvictorQueue::reverse_iterator(_queue.erase(mapPos->second->queuePos));
            _map.erase(mapPos);
        }
        else
        {
            ++p;
        }
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

C++11

```
void
EvictorBase::deactivate(const string&)
{
    lock_guard<mutex> lk(_mutex);

    _size = 0;
    evictServants();
}
```

Note that, with this implementation of `evictServants`, we only scan the tail section of the evictor queue for servants to evict. If we have long-running operations, this allows the number of servants in the queue to remain above the evictor size if the servants in the tail section have a non-zero use count. This means that, even immediately after calling `evictServants`, the queue length can still exceed the evictor size.

Using Servant Evictors in C++

Using a servant evictor is simply a matter of deriving a class from `EvictorBase` and implementing the `add` and `evict` methods. You can turn a servant locator into an evictor by simply taking the code that you wrote for `locate` and placing it into `add` — `EvictorBase` then takes care of maintaining the cache in least-recently used order and evicting servants as necessary. Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of `evict` can be left empty.

One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.

Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

See Also

- Servant Evictors
- Object Identity
- Default Servants

Implementing a Servant Evictor in Java

On this page:

The `EvictorBase` Class in Java

The evictor we show here is designed as an abstract base class: in order to use it, you derive a class from the `Evictor.EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definition as follows:

Java

```
package Evictor;

import com.zeroc.Ice.Current;
import com.zeroc.Ice.ServantLocator;

public abstract class EvictorBase implements ServantLocator
{
    public EvictorBase()
    {
        _size = 1000;
    }

    public EvictorBase(int size)
    {
        _size = size < 0 ? 1000 : size;
    }

    static public class AddResult
    {
        public com.zeroc.Ice.Object returnValue;
        public java.lang.Object cookie;
    }
    public abstract AddResult add(Current c);

    public abstract void evict(com.zeroc.Ice.Object servant, java.lang.
Object cookie);

    @Override
    synchronized public final ServantLocator.LocateResult locate(Curren
t c) { ... }

    @Override
    synchronized public final void finished(Current c, com.zeroc.Ice.Ob
ject o, java.lang.Object cookie) { ... }

    @Override
    synchronized public final void deactivate(String category) { ... }
    // ...

    private int _size;
}
```

Note that the evictor has constructors to set the size of the queue, with a default size of 1000.

The `locate`, `finished`, and `deactivate` methods are inherited from the `ServantLocator` interface; these methods implement the logic to maintain the queue in LRU order and to add and evict servants as needed. The methods are synchronized, so the evictor's internal data structures are protected from concurrent access.

The `add` and `evict` methods are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are abstract, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant, allowing the subclass to perform any cleanup. Note that `add` can also return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map not only stores servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The last two points deserve some extra explanation.

- The evictor queue must be maintained in least-recently used order, that is, every time an invocation arrives and we find an entry for the identity in the evictor map, we also must locate the servant's identity on the evictor queue and move it to the front of the queue. However, scanning for that entry is inefficient because it requires *O*(*n*) time. To get around this, we store an iterator in the evictor map that marks the corresponding entry's position in the evictor queue. This allows us to dequeue the entry from its current position and enqueue it at the head of the queue in *O*(1) time.

  Unfortunately, the various lists provided by `java.util` do not allow us to keep an iterator to a list position without invalidating that iterator as the list is updated. To deal with this, we use a special-purpose linked list implementation, `Evictor.LinkedList`, that does not have this limitation. `LinkedList` has an interface similar to `java.util.LinkedList` but does not invalidate iterators other than iterators that point at an element that is removed. For brevity, we do not show the implementation of this list here — you can find the implementation in the code examples for this manual in the Ice distribution.

- We maintain a use count as part of the map in order to avoid incorrect eviction of servants. Suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

  By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

  The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

Java

```
package Evictor;

import com.zeroc.Ice.Identity;
import com.zeroc.Ice.ServantLocator;

public abstract class EvictorBase implements ServantLocator
{
    // ...

    private class EvictorEntry
    {
        com.zeroc.Ice.Object servant;
        java.lang.Object userCookie;
        java.util.Iterator<Identity> queuePos;
        int useCount;
    }

    private void evictServants()
    {
        // ...
    }

    private java.util.Map<Identity, EvictorEntry> _map = new java.util.
HashMap<>();
    private Evictor.LinkedList<Identity> _queue = new Evictor.LinkedLis
t<>();
    private int _size;
}
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. The map key is the identity of the Ice object, and the lookup value is of type `EvictorEntry`. The queue simply stores identities, of type `Ice::Identity`.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit — we will discuss this function in more detail shortly.

Almost all the action of the evictor takes place in the implementation of `locate`:

Java

```
synchronized public final ServantLocator.LocateResult locate(Current c)
{
    ServantLocator.LocateResult r = new ServantLocator.LocateResult();
    //
    // Check if we have a servant in the map already.
    //
    EvictorEntry entry = _map.get(c.id);
    if(entry != null)
    {
        //
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        entry.queuePos.remove();
    }
    else
    {
        //
        // We do not have entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        entry = new EvictorEntry();
        AddResult ar = add(c); // Down-call
        if(ar.returnValue == null)
        {
            return r;
        }
        r.servant = entry.servant = ar.returnValue;
        entry.userCookie = ar.cookie;
        entry.useCount = 0;
        _map.put(c.id, entry);
    }

    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(entry.useCount);
    _queue.addFirst(c.id);
    entry.queuePos = _queue.iterator();
    entry.queuePos.next(); // Position iterator on the element.

    r.cookie = entry;
    return r;
}
```

The code uses an `EvictorEntry` as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding

call to `finished`.

We first look for an existing entry in the evictor map, using the object identity as the key. If we have an entry in the map already, we dequeue the corresponding identity from the evictor queue. (The `queuePos` member of `EvictorEntry` is an iterator that marks that entry's position in the evictor queue.)

Otherwise, we do not have an entry in the map, so we create a new one and call the `add` method. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either provide a servant in its result as usual, or provide a null servant or throw an exception to indicate failure. If `add` returns a null servant, we return a null servant to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The final few lines of code increment the entry's use count, add the entry at the head of the evictor queue, store the entry's position in the queue, and assign the entry to the cookie that is returned from `locate`, before returning the result to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

Java

```
synchronized public final void
finished(Current c, com.zeroc.Ice.Object o, java.lang.Object cookie)
{
    EvictorEntry entry = (EvictorEntry)cookie;

    // Decrement use count and check if
    // there is something to evict.
    //
    --(entry.useCount);
    evictServants();
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

Java

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    java.util.Iterator<Identity> p = _queue.riterator();
    int excessEntries = _map.size() - _size;
    for(int i = 0; i < excessEntries; ++i)
    {
        Identity id = p.next();
        EvictorEntry e = _map.get(id);
        if(e.useCount == 0)
        {
            evict(e.servant, e.userCookie); // Down-call
            e.queuePos.remove();
            _map.remove(id);
        }
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

Java

```
synchronized public final void deactivate(String category)
{
    _size = 0;
    evictServants();
}
```

Note that, with this implementation of `evictServants`, we only scan the tail section of the evictor queue for servants to evict. If we have long-running operations, this allows the number of servants in the queue to remain above the evictor size if the servants in the tail section have a non-zero use count. This means that, even immediately after calling `evictServants`, the queue length can still exceed the evictor size.

We can adopt a more aggressive strategy for eviction: instead of scanning only the excess entries in the queue, if, after looking in the tail section of the queue, we still have more servants in the queue than the queue size, we keep scanning for servants with a zero use count until the queue size drops below the limit. This alternative version of `evictServants` looks as follows:

Java

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    java.util.Iterator<Identity> p = _queue.riterator();
    int numEntries = _map.size();
    for(int i = 0; i < excessEntries && _map.size() > _size; ++i)
    {
        Identity id = p.next();
        EvictorEntry e = _map.get(id);
        if(e.useCount == 0)
        {
            evict(e.servant, e.userCookie); // Down-call
            e.queuePos.remove();
            _map.remove(id);
        }
    }
}
```

The only difference in this version is that the terminating condition for the `for`-loop has changed: instead of scanning only the excess entries for servants with a use count, this version keeps scanning until the evictor size drops below the limit.

Which version is more appropriate depends on your application: if locating and evicting servants is expensive, and memory is not at a premium, the first version (which only scans the tail section) is more appropriate; if you want to keep memory consumption to a minimum, the second version is more appropriate. Also keep in mind that the difference between the two versions is significant only if you have long-running operations and many concurrent invocations from clients; otherwise, there is no point in more aggressively scanning for servants to remove because they are going be become idle again very quickly and get evicted as soon as the next request arrives.

Using Servant Evictors in Java

Using a servant evictor is simply a matter of deriving a class from `EvictorBase` and implementing the `add` and `evict` methods. You can turn a servant locator into an evictor by simply taking the code that you wrote for `locate` and placing it into `add` — `EvictorBase` then takes care of maintaining the cache in least-recently used order and evicting servants as necessary. Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of `evict` can be left empty.

One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.

Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

See Also

- Servant Evictors
- Object Identity
- Default Servants

Implementing a Servant Evictor in C-Sharp

On this page:

A Linked List to Support Eviction in C#

The `System.Collections` classes do not provide a container that does not invalidate iterators when we modify the contents of the container but, to efficiently implement an evictor, we need such a container. To deal with this, we use a special-purpose linked list implementation, `Evictor.LinkedList`, that does not invalidate iterators when we delete or add an element. For brevity, we only show the interface of `LinkedList` here — you can find the implementation in the code examples for this manual in the Ice distribution.

C#

```
namespace Evictor
{
    public class LinkedList<T> : ICollection<T>, ICollection, ICloneable
    {
        public LinkedList();

        public int Count { get; }

        public void Add(T value);
        public void AddFirst(T value);
        public void Clear();
        public bool Contains(T value);
        public bool Remove(T value);

        public IEnumerator GetEnumerator();

        public class Enumerator : IEnumerator<T>, IEnumerator,
IDisposable
        {
            public void Reset();

            public T Current { get; }

            public bool MoveNext();
            public bool MovePrev();
            public void Remove();
            public void Dispose();
        }

        public void CopyTo(T[] array, int index);
        public void CopyTo(Array array, int index);

        public object Clone();

        public bool IsReadOnly { get; }
        public bool IsSynchronized { get; }
        public object SyncRoot { get; }
    }
}
```

The `Add` method appends an element to the list, and the `AddFirst` method prepends an element to the list. `GetEnumerator` returns an enumerator for the list elements; immediately after calling `GetEnumerator`, the enumerator does not point at any element until you call either `MoveNext` or `MovePrev`, which position the enumerator at the first and last element, respectively. `Current` returns the element at the enumerator position, and `Remove` deletes the element at the current position and leaves the enumerator pointing at no element. Calling `MoveNext` or `MovePrev` after calling `Remove` positions the enumerator at the element following or preceding the deleted element, respectively. `MoveNext` and `MovePrev` return true if they have positioned the enumerator on an element; otherwise, they return false and leave the enumerator position on the last and first element, respectively.

The `EvictorBase` Class in C#

Given this `LinkedList`, we can implement the evictor. The evictor we show here is designed as an abstract base class: in order to use it, you derive a class from the `Evictor.EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definition as follows:

C#

```
namespace Evictor
{
    public abstract class EvictorBase : Ice.ServantLocator
    {
        public EvictorBase()
        {
            _size = 1000;
        }

        public EvictorBase(int size)
        {
            _size = size < 0 ? 1000 : size;
        }

        protected abstract Ice.Object add(Ice.Current c,
out object cookie);

        protected abstract void evict(Ice.Object servant,
object cookie);

        public Ice.Object locate(Ice.Current c, out object cookie)
        {
            lock(this)
            {
                // ...
            }
        }

        public void finished(Ice.Current c, Ice.Object o, object cookie)
        {
            lock(this)
            {
                // ...
            }
        }

        public void deactivate(string category)
        {
            lock(this)
            {
                // ...
            }
        }

        private int _size;
    }
}
```

Note that the evictor has constructors to set the size of the queue, with a default size of 1000.

The `locate`, `finished`, and `deactivate` methods are inherited from the `ServantLocator` base class; these methods implement the logic to maintain the queue in LRU order and to add and evict servants as needed. The methods use a `lock(this)` statement for their body, so the evictor's internal data structures are protected from concurrent access.

The `add` and `evict` methods are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are abstract, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant, allowing the subclass to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map not only stores servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The last two points deserve some extra explanation.

- The evictor queue must be maintained in least-recently used order, that is, every time an invocation arrives and we find an entry for the identity in the evictor map, we also must locate the servant's identity on the evictor queue and move it to the front of the queue. However, scanning for that entry is inefficient because it requires *O*(*n*) time. To get around this, we store an iterator in the evictor map that marks the corresponding entry's position in the evictor queue. This allows us to dequeue the entry from its current position and enqueue it at the head of the queue in *O*(1) time, using the `Evictor.LinkedList` implementation.

- We maintain a use count as part of the map in order to avoid incorrect eviction of servants. Suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

  By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

  The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a reference to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

C#

```
namespace Evictor
{
    using System.Collections.Generic;

    public abstract class EvictorBase : Ice.ServantLocator
    {
        // ...

        private class EvictorEntry
        {
            internal Ice.Object servant;
            internal object userCookie;
            internal LinkedList<Ice.Identity>.Enumerator queuePos;
            internal int useCount;
        }

        private void evictServants()
        {
            // ...
        }

        private Dictionary<Ice.Identity, EvictorEntry> _map =
            new Dictionary<Ice.Identity, EvictorEntry>();
        private LinkedList<Ice.Identity> _queue =
            new LinkedList<Ice.Identity>();
        private int _size;
    }
}
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. The map key is the identity of the Ice object, and the lookup value is of type `EvictorEntry`. The queue simply stores identities, of type `Ice.Identity`.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit — we will discuss this function in more detail shortly.

Almost all the action of the evictor takes place in the implementation of `locate`:

C#

```
public Ice.Object locate(Ice.Current c, out object cookie)
{
    lock(this)
    {
        //
        // Check if we a servant in the map already.
        //
        EvictorEntry entry = _map[c.id];
        if(entry != null)
        {
            //
            // Got an entry already, dequeue the entry from
            // its current position.
            //
            entry.queuePos.Remove();
        }
        else
        {
            //
            // We do not have an entry. Ask the derived class to
            // instantiate a servant and add an entry to the map.
            //
            entry = new EvictorEntry();
            entry.servant = add(c, out entry.userCookie);
            if(entry.servant == null)
            {
                cookie = null;
                return null;
            }
            entry.useCount = 0;
            _map[c.id] = entry;
        }

        //
        // Increment the use count of the servant and enqueue
        // the entry at the front, so we get LRU order.
        //
        ++(entry.useCount);
        _queue.AddFirst(c.id);
        entry.queuePos = (LinkedList<Ice.Identity>.Enumerator)_queue.Ge
tEnumerator();
        entry.queuePos.MoveNext();

        cookie = entry;

        return entry.servant;
    }
}
```

The code uses an `EvictorEntry` as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding

call to `finished`.

We first look for an existing entry in the evictor map, using the object identity as the key. If we have an entry in the map already, we dequeue the corresponding identity from the evictor queue. (The `queuePos` member of `EvictorEntry` is an iterator that marks that entry's position in the evictor queue.)

Otherwise, we do not have an entry in the map, so we create a new one and call the `add` method. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we return null to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The final few lines of code increment the entry's use count, add the entry at the head of the evictor queue, store the entry's position in the queue, and initialize the cookie that is returned from `locate`, before returning the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

C#

```
public void finished(Ice.Current c, Ice.Object o, object cookie)
{
    lock(this)
    {
        EvictorEntry entry = (EvictorEntry)cookie;

        //
        // Decrement use count and check if
        // there is something to evict.
        //
        --(entry.useCount);
        evictServants();
    }
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

C#

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    LinkedList<Ice.Identity>.Enumerator p =
        (LinkedList<Ice.Identity>.Enumerator)_queue.GetEnumerator();
    int excessEntries = _map.Count - _size;
    for(int i = 0; i < excessEntries; ++i)
    {
        p.MovePrev();
        Ice.Identity id = p.Current;
        EvictorEntry e = _map[id];
        if(e.useCount == 0)
        {
            evict(e.servant, e.userCookie); // Down-call
            p.Remove();
            _map.Remove(id);
        }
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

C#

```
public void deactivate(string category)
{
    lock(this)
    {
        _size = 0;
        evictServants();
    }
}
```

Note that, with this implementation of `evictServants`, we only scan the tail section of the evictor queue for servants to evict. If we have long-running operations, this allows the number of servants in the queue to remain above the evictor size if the servants in the tail section have a non-zero use count. This means that, even immediately after calling `evictServants`, the queue length can still exceed the evictor size.

We can adopt a more aggressive strategy for eviction: instead of scanning only the excess entries in the queue, if, after looking in the tail section of the queue, we still have more servants in the queue than the queue size, we keep scanning for servants with a zero use count

until the queue size drops below the limit. This alternative version of `evictServants` looks as follows:

C#

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    LinkedList<Ice.Identity>.Enumerator p =
        (LinkedList<Ice.Identity>.Enumerator)_queue.GetEnumerator();
    int numEntries = _map.Count;
    for(int i = 0; i < numEntries && _map.Count > _size; ++i)
    {
        p.MovePrev();
        Ice.Identity id = p.Current;
        EvictorEntry e = _map[id];
        if(e.useCount == 0)
        {
            evict(e.servant, e.userCookie); // Down-call
            p.Remove();
            _map.Remove(id);
        }
    }
}
```

The only difference in this version is that the terminating condition for the `for`-loop has changed: instead of scanning only the excess entries for servants with a use count, this version keeps scanning until the evictor size drops below the limit.

Which version is more appropriate depends on your application: if locating and evicting servants is expensive, and memory is not at a premium, the first version (which only scans the tail section) is more appropriate; if you want to keep memory consumption to a minimum, the second version is more appropriate. Also keep in mind that the difference between the two versions is significant only if you have long-running operations and many concurrent invocations from clients; otherwise, there is no point in more aggressively scanning for servants to remove because they are going be become idle again very quickly and get evicted as soon as the next request arrives.

Using Servant Evictors in C#

Using a servant evictor is simply a matter of deriving a class from `EvictorBase` and implementing the `add` and `evict` methods. You can turn a servant locator into an evictor by simply taking the code that you wrote for `locate` and placing it into `add` — `EvictorBase` then takes care of maintaining the cache in least-recently used order and evicting servants as necessary. Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of `evict` can be left empty.

One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.

Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

See Also

- Servant Evictors
- Object Identity

- Default Servants

Object Life Cycle

Topics

Understanding Object Life Cycle

Object life cycle generally refers to how an object-oriented application (whether distributed or not) creates and destroys objects. For distributed applications, life cycle management presents particular challenges. For example, destruction of objects often can be surprisingly complex, especially in threaded applications. Before we go into the details of object creation and destruction, we need to have a closer look what we mean by the terms "life cycle" and "object" in this context.

Object life cycle refers to the act of creation and destruction of objects. For example, with our file system application, we may start out with an empty file system that only contains a root directory. Over time, clients (by as yet unspecified means) add new directories and files to the file system. For example, a client might create a new directory called `MyPoems` underneath the root directory. Some time later, the same or a different client might decide to remove this directory again, returning the file system to its previous empty state. This pattern of creation and destruction is known as object life cycle.

The life cycle of distributed objects raises a number of interesting and challenging questions. For example, what should happen if a client destroys a file while another client is reading or writing that file? And how do we prevent two files with the same name from existing in the same directory? Another interesting scenario is illustrated by the following sequence of events:

1. Client A creates a file called `DraftPoem` in the root directory and uses it for a while.
2. Some time later, client B destroys the `DraftPoem` file so it no longer exists.
3. Some time later still, client C creates a new `DraftPoem` file in the root directory, with different contents.
4. Finally, client A attempts to access the `DraftPoem` file it created earlier.

What should happen when, in the final step, client A tries to use the `DraftPoem` file? Should the client's attempt succeed and simply operate on the new contents of the file that were placed there by client C? Or should client A's attempt fail because, after all, the new `DraftPoem` file is, in a sense, a completely different file from the original one, even though it has the same name?

The answers to such questions cannot be made in general. Instead, meaningful answers depend on the semantics that each individual application attaches to object life cycle. In this chapter, we will explore the various possible interpretations and how to implement them correctly, particularly for threaded applications.

See Also

- Slice for a Simple File System

Object Existence and Non-Existence

Before we talk about how to create and destroy objects, we need to look at a more basic concept, namely that of object existence. What does it mean for an object to "exist" and, more fundamentally, what do we mean by the term "object"?

As mentioned in Ice Objects, an *Ice object* is a conceptual entity, or abstraction that does not really exist. On the client side, the concrete representation of an Ice object is a proxy and, on the server side, the concrete representation of an Ice object is a servant. Proxies and servants are the concrete programming-language artifacts that represent Ice objects.

Because Ice objects are abstract, conceptual entities, they are invisible to the Ice run time and to the application code — only proxies and servants are real and visible. It follows that, to determine whether an *Ice* object exists, any determination must rely on proxies and servants, because they are the only tangible entities in the system.

On this page:

Object Non-Existence

Here is the definitive statement of what it means for an Ice object to *not* exist:

> *An Ice object does not exist if an invocation on the object raises an `ObjectNotExistException`.*

This may seem self-evident but, on closer examination, is a little more subtle than you might expect. In particular, Ice object existence has meaning *only within the context of a particular invocation*. If that invocation raises `ObjectNotExistException`, the object is known to not exist. Note that this says nothing about whether concurrent or future requests to that object will also raise `ObjectNotExistException` — they may or may not, depending on the semantics that are implemented by the application.

Also note that, because all the Ice run time knows about are servants, an `ObjectNotExistException` really indicates that a *servant* for the request could not be found at the time the request was made. This means that, ultimately, it is the application that attaches the meaning "the Ice object does not exist" to this exception.

In theory, the application can attach any meaning it likes to `ObjectNotExistException` and a server can throw this exception for whatever reason it sees fit; in practice, however, we recommend that you do not do this because it breaks with existing convention and is potentially confusing. You should reserve this exception for its intended meaning and not abuse it for other purposes.

Object Existence

The preceding definition does not say anything about object existence if something other than `ObjectNotExistException` is returned in response to a particular request. So, here is the definitive statement of what it means for an Ice object to exist:

> *An Ice object exists if a twoway invocation on the object either succeeds, raises a user exception, or raises `FacetNotExistException` or `OperationNotExistException`.*

It is self-evident that an Ice object exists if a twoway invocation on it succeeds: obviously, the object received the invocation, processed it, and returned a result. However, note the qualification: this is true only for *twoway* invocations; for oneway and datagram invocations, nothing can be inferred about the existence of the corresponding Ice object by invoking an operation on it: because there is no reply from the server, the client-side Ice run time has no idea whether the request was dispatched successfully in the server or not. This includes user exceptions, `ObjectNotExistException`, `FacetNotExistException`, and `OperationNotExistException` — these exceptions are never raised by oneway and datagram invocations, regardless of the actual state of the target object.

If a twoway invocation raises a user exception, the Ice object obviously exists: the Ice run time never raises user exceptions so, for an invocation to raise a user exception, the invocation was dispatched successfully in the server, and the operation implementation in the servant raised the exception.

If a twoway invocation raises `FacetNotExistException`, we do know that the corresponding Ice object indeed exists: the Ice run time raises `FacetNotExistException` only if it can find the identity of the target object in the Active Servant Map (ASM), but cannot find the facet that was specified by the client.

> Note that, if you use servant locators, for these semantics to hold, your servant locator must correctly raise `FacetNotExistException` in the `locate` operation (instead of returning null or raising `ObjectNotExistException`) if an Ice object exists, but the particular target facet does not exist.

As a corollary to the preceding two definitions, we can state:

> *A facet does not exist if a twoway invocation on the object raises `ObjectNotExistException` or `FacetNotExistException`.*
>
> *A facet exists if a twoway invocation on the object either succeeds, or raises `OperationNotExistException`.*

These definitions simply capture the fact that a facet is a "sub-object" of an Ice object: if an invocation raises `ObjectNotExistException`, we know that the facet does not exist either because, for a facet to exist, its Ice object must exist.

If an operation raises `OperationNotExistException`, we know that both the target Ice object and the target facet exist. However, the operation that the client attempted to invoke does not. (This is possible only if you use dynamic invocation or if you have mis-matched Slice definitions for client and server.)

Indeterminate Object State

The preceding definitions clearly state under what circumstances we can conclude that an Ice object (or its facet) does or does not exist. However, the preceding definitions are incomplete because operation invocations can have outcomes other than success or failure with `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException`. For example, a client might receive a `MarshalException`, `UnknownLocalException`, `UnknownException`, or `TimeoutException`. In that case, the client cannot draw any conclusions about whether the Ice object on which it invoked a twoway operation exists or not — the exceptions simply indicate that something went wrong while the invocation was processed. So, to complete our definitions, we can state:

> *If a twoway invocation raises a run-time exception other than `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException`, nothing is known about the existence or non-existence of the Ice object that was the target of the invocation. Furthermore, it is impossible to determine the state of existence of an Ice object with a oneway or datagram invocation.*

Authoritative Object Existence Semantics

The preceding definitions capture the fact that, to make a determination of object existence or non-existence, the client-side Ice run time must be able to contact the server and, moreover, receive a reply from the server:

- If the server can be contacted and returns a successful reply for an invocation, the Ice object exists.
- If the server can be contacted and returns an `ObjectNotExistException` (or `FacetNotExistException`), the Ice object (or facet) does not exist. If the server returns an `OperationNotExistException`, the Ice object (and its facet) exists, but does not provide the requested operation, which indicates a type mismatch due to client and server using out-of-sync Slice definitions or due to incorrect use of dynamic invocation.
- If the server cannot be contacted, does not return a reply (as for oneway and datagram invocations), or if anything at all goes wrong with the process of sending an invocation, processing it in the server, and returning the reply, nothing is known about the state of the Ice object, including its existence or non-existence.

Another way of looking at this is that a decision as to whether an object exists or not is *never* made by the Ice run time and, instead, is *always* made by the server-side *application* code:

- If an invocation completes successfully, the server-side application code was involved because it processed the invocation.
- If an invocation returns `ObjectNotExistException` or `FacetNotExistException`, the server-side application code was also involved:
  - either the Ice run time could not find a servant for the invocation in the ASM, in which case the application code was involved by virtue of not having added a servant to the ASM in the first place, or
  - the Ice run time consulted a servant locator that explicitly returned null or raised `ObjectNotExistException` or `FacetNotExistException`.

This means that `ObjectNotExistException` and `FacetNotExistException` are *authoritative*: when you receive these exceptions, you can always believe what they tell you — the Ice run time never raises these exceptions without consulting your code, either implicitly (via an ASM lookup) or explicitly (by calling a servant locator's `locate` operation).

These semantics are motivated by the need to keep the Ice run time stateless with respect to object existence. For example, it would be nice to have stronger semantics, such as a promise that "once an Ice object has existed and been destroyed, all future requests to that Ice object also raise `ObjectNotExistException`". However, to implement these semantics, the Ice run time would have to remember all object identities that were used in the past, and prevent their reuse for new Ice objects. Of course, this would be inherently non-scalable. In addition, it would prevent applications from controlling object identity; allowing such control for applications is important however, for example, to link the identity of an Ice object to its persistent state in a database.

Note that, if the implementation of an operation calls another operation, dealing with `ObjectNotExistException` may require some care. For example, suppose that the client holds a proxy to an object of type `Service` and invokes an operation `provideService` on it:

C++11

```
shared_ptr<ServicePrx> service = ...;

try
{
    service->provideService();
}
catch(const ObjectNotExistException&)
{
    // Service does not exist.
}
```

Here is the implementation of `provideService` in the server, which makes a call on a helper object to implement the operation:

C++11

```
void
ServiceI::provideService(const Ice::Current&)
{
    // ...
    proxyToHelper->someOp();
    // ...
}
```

If `proxyToHelper` happens to point at an object that was destroyed previously, the call to `someOp` will throw `ObjectNotExistException`. If the implementation of `provideService` does not intercept this exception, the exception will propagate all the way back to the client, who will conclude that the service has been destroyed when, in fact, the service still exists but the helper object used to implement `provideService` no longer exists.

Usually, this scenario is not a serious problem. Most often, the helper object cannot be destroyed while it is needed by `provideService` due to the way the application is structured. In that case, no special action is necessary because `someOp` will never throw `ObjectNotExistException`. On the other hand, if it is possible for the helper object to be destroyed, `provideService` can wrap a try-catch block for `ObjectNotExistException` around the call to `someOp` and throw an appropriate user exception from the exception handler (such as `ResourceUnavailable` or similar).

See Also

- Terminology
- Servant Locators
- Oneway Invocations
- Datagram Invocations
- Versioning
- Dynamic Ice

Life Cycle of Proxies, Servants, and Ice Objects

It is important to be aware of the different roles of proxies, servants, and Ice objects in a system. Proxies are the client-side representation of Ice objects and servants are the server-side representation of Ice objects. Proxies, servants, and Ice objects have completely independent life cycles. Clients can create and destroy proxies with or without a corresponding servant or Ice object in existence, servers can create and destroy servants with or without a corresponding proxy or Ice object in existence and, most importantly, Ice objects can exist or not exist regardless of whether corresponding proxies or servants exist. Here are a few examples to illustrate this:

C++11

```
{
    auto obj = communicator->stringToProxy("hello:tcp -p 10000");
    // Proxy exists now.

} // Proxy ceases to exist.
```

This code creates a proxy to an Ice object with the identity `Hello`. The server for this Ice object is expected to listen for invocations on the same host as the client, on port 10000, using the TCP/IP protocol. The proxy exists as soon as the call to `stringToProxy` completes and, thereafter, can by used by the client to make invocations on the corresponding Ice object.

However, note that this code says nothing at all about whether or not the corresponding Ice object exists. In particular, there might not be any Ice object with the identity `Hello`. Or there might be such an object, but the server for it may be down or unreachable. It is only when the client makes an invocation on the proxy that we get to find out whether the object exists, does not exist, or cannot be reached.

Similarly, at the end of the scope enclosing the `obj` variable in the preceding code, the proxy goes out of scope and is destroyed. Again, this says nothing about the state of the corresponding Ice object or its servant. This shows that the life cycle of a proxy is completely independent of the life cycle of its Ice object and the servant for that Ice object: clients can create and destroy proxies whenever they feel like it, and doing so has no implications for Ice objects or servant creation or destruction.

Here is another code example, this time for the server side:

C++11

```
{
    auto file = make_shared<FileI>("DraftPoem", root);
    // Servant exists now.

} // Servant ceases to exist.
```

Here, the server instantiates a servant for a File object by creating a `FileI` instance. The servant comes into being as soon as the call to `new` completes and ceases to exist as soon as the scope enclosing the `file` variable closes. Note that, as for proxies, the life cycle of the servant is completely independent of the life cycle of proxies and Ice objects. Clearly, the server can create and destroy a servant regardless of whether there are any proxies in existence for the corresponding Ice object.

Similarly, an Ice object can exist even if no servants exist for it. For example, our Ice objects might be persistent and stored in a database; in that case, if we switch off the server for our Ice objects, no servants exist for these Ice objects, even though the Ice objects continue to exist — the Ice objects are temporarily inaccessible, but exist regardless and, once their server is restarted, will become accessible again.

Conversely, a servant can exist without its corresponding Ice object. The mere creation of a servant does nothing, as far as the Ice run time is concerned. It is only once a servant is added to the Active Servant Map (ASM) (or a servant locator returns the servant from `locate`) that the servant incarnates its Ice object.

Finally, an Ice object can exist independently of proxies and servants. For example, returning to the database example, we might have an Ice server that acts as a front end to an online telephone book: each entry in the phone book corresponds to a separate Ice object. When a client invokes an operation, the server uses the identity of the incoming request to determine which Ice object is the target of the request and then contacts the back-end database to, for example, return the street address of the entry. With such a design, entries can be added to and removed from the back-end database quite independently of what happens to proxies and servants — the server finds out whether an Ice object exists only when it accesses the back-end database.

The only time that the life cycle of an Ice object and a servant are linked is during an invocation on that Ice object: for an invocation to

complete successfully, a servant must exist for the *duration of the invocation*. What happens to the servant thereafter is irrelevant to clients and, in general, is irrelevant to the corresponding Ice object.

It is important to be clear about the independence of the life cycles of proxies, servants, and Ice objects because this independence has profound implications for how you need to implement object life cycle. In particular, to destroy an Ice object, a client cannot simply destroy its proxy for an object because the server is completely unaware when a client does this.

See Also

- The Active Servant Map
- Servant Locators
- Object Identity

Object Creation

Now that we understand what it means for an Ice object to exist, we can look at what is involved in creating an Ice object. Fundamentally, there is only one way for an Ice object to come into being: the server must instantiate a servant for the object and add an entry for that servant to the Active Servant Map (ASM) (or, alternatively, arrange for a servant locator to return a servant from its `locate` operation).

> For the remainder of this chapter, we will ignore the distinction between using the ASM and a servant locator and simply assume that the code uses the ASM. This is because servant locators do not alter the discussion: if `locate` returns a servant, that is the same as a successful lookup in the ASM; if `locate` returns null or throws `ObjectNotExistException`, that is the same as an unsuccessful lookup in the ASM.

One obvious way for a server to create a servant is to, well, simply instantiate it and add it to the ASM of its own accord. For example:

C++11

```
auto root = make_shared<DirectoryI>("/", 0);
adapter->addWithUUID(root); // Ice object exists now
```

The servant exists as soon as the call to `new` completes, and the Ice object exists as soon as the code adds the servant to the ASM: at that point, the Ice object becomes reachable to clients who hold a proxy to it.

This is the way we created Ice objects for our file system application earlier in the manual. However, doing so is not all that interesting because the only files and directories that exist are those that the server decides to create when it starts up. What we really want is a way for *clients* to create and destroy directories and files.

On this page:

Creating an Object with a Factory

The canonical way to create an object is to use the factory pattern [1]. The factory pattern, in a nutshell, says that objects are created by invoking an operation (usually called `create`) on an object factory:

Slice

```
interface PhoneEntry
{
    idempotent string name();
    idempotent string getNumber();
    idempotent void setNumber(string phNum);
}

exception PhoneEntryExists
{
    string name;
    string phNum;
}

interface PhoneEntryFactory
{
    PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;
}
```

> Rather than continue with the file system example, we will simplify the discussion for the time being by using the phone book example mentioned earlier; we will return to the file system application to explore more complex issues.

The entries in the phone book consist of simple name-number pairs. The interface to each entry is called `PhoneEntry` and provides operations to read the name and to read and write the phone number. (For a real application, the objects would likely be more complex and encapsulate more state. However, these simple objects will do for the purposes of this discussion.)

To create a new entry, a client calls the `create` operation on a `PhoneEntryFactory` object. (The factory is a singleton object [1] — that is, only one instance of that interface exists in the server.) It is the job of `create` to create a new `PhoneEntry` object, using the supplied name as the object identity.

An immediate consequence of using the name as the object identity is that `create` can raise a `PhoneEntryExists` exception: presumably, if a client attempts to create an entry with the same name as an already-existing entry, we need to let the client know about this. (Whether this is an appropriate design is something we examine more closely in Object Identity and Uniqueness.)

`create` returns a proxy to the newly-created object, so the client can use that proxy to invoke operations. However, this is by convention only. For example, `create` could be a `void` operation if the client has some other way to eventually get a proxy to the new object (such as creating the proxy from a string, or locating the proxy via a search operation). Alternatively, you could define "bulk" creation operations that allow clients to create several new objects with a single RPC. As far as the Ice run time is concerned, there is nothing special about a factory operation: a factory operation is just like any other operation; it just so happens that a factory operation creates a new Ice object as a side effect of being called, that is, the *implementation* of the operation is what creates the object, not the Ice run time.

Also note that `create` accepts a `name` and a `phNum` parameter, so it can initialize the new object. This is not compulsory, but generally a good idea. An alternate factory operation could be:

Slice

```
interface PhoneEntryFactory
{
    PhoneEntry* create(string name)
        throws PhoneEntryExists;
}
```

With this design, the assumption is that the client will call `setNumber` after it has created the object. However, in general, allowing objects that are not fully initialized is a bad idea: it all too easily happens that a client either forgets to complete the initialization, or happens to crash or get disconnected before it can complete the initialization. Either way, we end up with a partially-initialized object in the system that can cause surprises later.

Similarly, so-called `generic` factories are also something to be avoided:

Slice

```
dictionary<string, string> Params;

exception CannotCreateException
{
    string reason;
}

interface GenericFactory
{
    Object* create(Params p)
        throws CannotCreateException;
}
```

The intent here is that a `GenericFactory` can be used to create any kind of object; the `Params` dictionary allows an arbitrary number of parameters to be passed to the `create` operation in the form of name — value pairs, for example:

C++11

```
shared_ptr<GenericFactoryPrx> factory = ...;

shared_ptr<Ice::ObjectPrx> obj;
Params p;

// Make a car.
//
p["Make"] = "Ford";
p["Model"] = "Falcon";
obj = factory->create(p);
auto car = Ice::checkedCast<CarPrx>(obj);

// Make a horse.
//
p.clear();
p["Breed"] = "Clydesdale";
p["Sex"] = "Male";
obj = factory->create(p);
auto horse = Ice::checkedCast<HorsePrx>(obj);
```

We strongly discourage you from creating factory interfaces such as this, unless you have a good overriding reason: generic factories undermine type safety and are much more error-prone than strongly-typed factories.

Implementing a Factory Operation

The implementation of an object factory is simplicity itself. Here is how we could implement the `create` operation for our `PhoneEntryFactory`:

C++11

```
shared_ptr<PhoneEntryPrx>
PhoneEntryFactory::create(string name, string phNum, const Current& c)
{
    try
    {
        auto comm = c.adapter->getCommunicator();
        auto servant = make_shared<PhoneEntryI>(name, phNum);
        return Ice::uncheckedCast<PhoneEntryPrx>(c.adapter->add(servant,
Ice::stringToIdentity(name)));
    }
    catch(const Ice::AlreadyRegisteredException&)
    {
        throw PhoneEntryExists(name, phNum);
    }
}
```

The `create` function instantiates a new `PhoneEntryI` object (which is the servant for the new `PhoneEntry` object), adds the servant to the ASM, and returns the proxy for the new object. Adding the servant to the ASM is what creates the new Ice object, and client requests are dispatched to the new object as soon as that entry appears in the ASM (assuming the object adapter is active).

Note that, even though this code contains no explicit lock, it is thread-safe. The `add` operation on the object adapter is atomic: if two clients concurrently add a servant with the same identity, exactly one thread succeeds in adding the entry to the ASM; the other thread receives an `AlreadyRegisteredException`. Similarly, if two clients concurrently call `create` for different entries, the two calls execute concurrently in the server (if the server is multi-threaded); the implementation of `add` in the Ice run time uses appropriate locks to ensure that concurrent updates to the ASM cannot corrupt anything.

See Also

- The Active Servant Map
- Servant Locators
- Object Identity
- Object Identity and Uniqueness
- Object Life Cycle for the File System Application
- Object Adapter States
- Servant Activation and Deactivation

References

1. Gamma, E., et al. 1994. Design Patterns. Reading, MA: Addison-Wesley.

Object Destruction

Now that clients can create `PhoneEntry` objects, let us consider how to allow clients to destroy them again. One obvious design is to add a `destroy` operation to the factory — after all, seeing that a factory knows how to create objects, it stands to reason that it also knows how to destroy them again:

Slice

```
exception PhoneEntryNotExists
{
    string name;
    string phNum;
}

interface PhoneEntryFactory
{
    PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;
    void destroy(PhoneEntry* pe)      // Bad idea!
        throws PhoneEntryNotExists;
}
```

While this works (and certainly can be implemented without problems), it is generally a bad idea. For one, an immediate problem we need to deal with is what should happen if a client passes a proxy to an already-destroyed object to `destroy`. We could raise an `ObjectNotExistException` to indicate this, but that is not a good idea because it makes it ambiguous as to which object does not exist: the factory, or the entry. (By convention, if a client receives an `ObjectNotExistException` for an invocation, what does not exist is the object the operation was targeted at, not some other object that in turn might be contacted by the operation.) This forces us to add a separate `PhoneEntryNotExists` exception to deal with the error condition, which makes the interface a little more complex.

A second and more serious problem with this design is that, in order to destroy an entry, the client must not only know which entry to destroy, but must also know *which factory created the entry*. For our example, with only a single factory, this is not a real concern. However, for more complex systems with dozens of factories (possibly in multiple server processes), it rapidly becomes a problem: for each object, the application code somehow has to keep track of which factory created what object; if any part of the code ever loses track of where an object originally came from, it can no longer destroy that object.

Of course, we could mitigate the problem by adding an operation to the `PhoneEntry` interface that returns a proxy to its factory. That way, clients could ask each object to provide the factory that created the object. However, that needlessly complicates the Slice definitions and really is just a band-aid on a fundamentally flawed design. A much better choice is to add the `destroy` operation to the `PhoneEntry` interface instead:

Slice

```
interface PhoneEntry
{
    idempotent string name();
    idempotent string getNumber();
    idempotent void setNumber(string phNum);
    void destroy();
}
```

With this approach, there is no need for clients to somehow keep track of which factory created what object. Instead, given a proxy to a `PhoneEntry` object, a client simply invokes the `destroy` operation on the object and the `PhoneEntry` obligingly commits suicide. Note that we also no longer need a separate exception to indicate the "object does not exist" condition because we can raise `ObjectNotExistException` instead — the exception exists precisely to indicate this condition and, because `destroy` is now an operation on the phone entry itself, there is no ambiguity about which object it is that does not exist.

Topics

- Idempotency and Life Cycle Operations
- Implementing a destroy Operation
- Cleaning Up a Destroyed Servant
- Life Cycle and Collection Operations
- Life Cycle and Normal Operations

See Also

- Object Creation

Idempotency and Life Cycle Operations

You may be tempted to write the life cycle operations as follows:

Slice

```
interface PhoneEntry
{
    // ...
    idempotent void destroy(); // Wrong!
}

interface PhoneEntryFactory
{
    idempotent PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;
}
```

The idea is that `create` and `destroy` can be idempotent operations because it is safe to let the Ice run time retry the operation in the event of a temporary network failure. However, this assumption is not true. To see why, consider the following scenario:

1. A client invokes `destroy` on a phone entry.
2. The Ice run time sends the request to the server on the wire.
3. The connection goes down just after the request was sent, but before the reply for the request arrives in the client. It so happens that the request was received by the server and acted upon, and the reply from the server back to the client is lost because the connection is still down.
4. The Ice run time tries to read the reply for the request and realizes that the connection has gone down. Because the operation is marked idempotent, the run time attempts an automatic retry by re-establishing the connection and sending the request a second time, which happens to work.
5. The server receives the request to destroy the entry but, because the entry is destroyed already, the server returns an `ObjectNotExistException` to the client, which the Ice run time passes to the application code.
6. The application receives an `ObjectNotExistException` and falsely concludes that it tried to destroy a non-existent object when, in fact, the object did exist and was destroyed as intended.

A similar scenario can be constructed for `create`: in that case, the application will receive a `PhoneEntryExists` exception when, in fact, the entry did not exist and was created successfully.

These scenarios illustrate that `create` and `destroy` are *never* idempotent: sending one `create` or `destroy` invocation for a particular object is not the same as sending two invocations: the outcome depends on whether the first invocation succeeded or not, so `create` and `destroy` are not idempotent.

See Also

- Idempotent Operations
- Automatic Retries

Implementing a destroy Operation

As far as the Ice run time is concerned, the act of destroying an Ice object is to remove the mapping between its proxy and its servant. In other words, an Ice object is destroyed when we remove its entry from the Active Servant Map (ASM). Once the ASM entry is gone, incoming operations for the object raise `ObjectNotExistException`, as they should.

On this page:

Object Destruction and Concurrency

Here is the simplest version of `destroy`:

C++11

```
void
PhoneEntryI::destroy(const Current& c)
{
    try
    {
        c.adapter->remove(c.id);
    }
    catch(const Ice::NotRegisteredException&)
    {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

The implementation removes the ASM entry for the servant, thereby destroying the Ice object. If the entry does not exist (presumably, because the object was destroyed previously), `destroy` throws an `ObjectNotExistException`, as you would expect.

The ASM entry is removed as soon as `destroy` calls `remove` on the object adapter. Assuming that we implement `create` as we saw earlier, so no other part of the code retains a smart pointer to the servant, this means that the ASM holds the only smart pointer to the servant, so the servant's reference count is 1.

Once the ASM entry is removed (and its smart pointer destroyed), the reference count of the servant drops to zero. In C++, this triggers a call to the destructor of the servant, and the heap-allocated servant is deleted just as it should be; in languages such as Java and C#, this makes the servant eligible for garbage collection, so it will be deleted eventually as well.

Things get more interesting if we consider concurrent scenarios. One such scenario involves concurrent calls to `create` and `destroy`. Suppose we have the following sequence of events:

1. Client A creates a phone entry.
2. Client A passes the proxy for the entry to client B.
3. Client A destroys the entry.
4. Client A calls `create` for the same entry (passing the same name, which serves as the object identity) and, concurrently, client B calls `destroy` on the entry.

Clearly, something is strange about this scenario, because it involves two clients asking for conflicting things, with one client trying to create an object that existed previously, while another client tries to destroy the object that — unbeknownst to that client — was destroyed earlier.

Exactly what is seen by client A and client B depends on how the operations are dispatched in the server. In particular, the outcome depends on the order in which the calls on the object adapter to `add` (in `create`) and `remove` (in `destroy`) on the servant are executed:

- If the thread processing client A's invocation executes `add` before the thread processing client B's invocation, client A's call to `add` s ucceeds. Internally, the calls to `add` and `remove` are serialized, and client B's call to `remove` blocks until client A's call to `add` has completed. The net effect is that both clients see their respective invocations complete successfully.
- If the thread processing client B's invocation executes `remove` before the thread processing client A's invocation executes `add`,

client B's thread receives a `NotRegisteredException`, which results in an `ObjectNotExistException` in client B. Client A's thread then successfully calls `add`, creating the object and returning its proxy.

This example illustrates that, if life cycle operations interleave in this way, the outcome depends on thread scheduling. However, as far as the Ice run time is concerned, doing this is perfectly safe: concurrent access does not cause problems for memory management or the integrity of data structures.

The preceding scenario allows two clients to attempt to perform conflicting operations. This is possible because clients can control the object identity of each phone entry: if the object identity were hidden from clients and assigned by the server (the server could assign a UUID to each entry, for example), the above scenario would not be possible. We will return to a more detailed discussion of such object identity issues in Object Identity and Uniqueness.

Concurrent Execution of Life Cycle and Non-Life Cycle Operations

> This section applies to C++ only.

Another scenario relates to concurrent execution of ordinary (non-life cycle) operations and `destroy`:

- Client A holds a proxy to an existing object and passes that proxy to client B.
- Client B calls the `setNumber` operation on the object.
- Client A calls `destroy` on the object *while Client B's call to `setNumber` is still executing*.

The immediate question is what this means with respect to memory management. In particular, client A's thread calls `remove` on the object adapter while client B's thread is still executing inside the object. If this call to `remove` were to delete the servant immediately, it would delete the servant while client B's thread is still executing inside the servant, with potentially disastrous results.

The answer is that this cannot happen. Whenever the Ice run time dispatches an incoming invocation to a servant, it increments the servant's reference count for the duration of the call, and decrements the reference count again once the call completes. Here is what happens to the servant's reference count for the preceding scenario:

1. Initially, the servant is idle, so its reference count is at least 1 because the ASM entry stores a smart pointer to the servant. (The remainder of these steps assumes that the ASM stores the *only* smart pointer to the servant, so the reference count is exactly 1.)
2. Client B's invocation of `setNumber` arrives and the Ice run time increments the reference count to 2 before dispatching the call.
3. While `setNumber` is still executing, client A's invocation of `destroy` arrives and the Ice run time increments the reference count to 3 before dispatching the call.
4. Client A's thread calls `remove` on the object adapter, which destroys the smart pointer in the ASM and so decrements the reference to 2.
5. Either `setNumber` or `destroy` may complete first. It does not matter which call completes — either way, the Ice run time decrements the reference count as the call completes, so after one of these calls completes, the reference count drops to 1.
6. Eventually, when the final call (`setNumber` or `destroy`) completes, the Ice run time decrements the reference count once again, which causes the count to drop to zero. In turn, this triggers the call to `delete` (which calls the servant's destructor).

The net effect is that, while operations are executing inside a servant, the servant's reference count is always greater than zero. As the invocations complete, the reference count drops until, eventually, it reaches zero. However, that can only happen once no operations are executing, that is, once the servant is idle. This means that the Ice run time guarantees that a servant's destructor runs only once the final operation invocation has drained out of the servant, so it is impossible to "pull memory out from underneath an executing invocation".

> For garbage-collected languages, such as C# and Java, the language run time provides the same semantics: while the servant can be reached via any reference in the application or the Ice run time, the servant will not be reclaimed by the garbage collector.

See Also

- The Active Servant Map
- Object Identity and Uniqueness
- Servant Activation and Deactivation

Cleaning Up a Destroyed Servant

Here is a very simple implementation of our `PhoneEntryI` servant. (Methods are inlined for convenience only. Also note that, for the time being, this code ignores concurrency issues, which we return to in Life Cycle and Normal Operations.)

C++11

```
class PhoneEntryI : public PhoneEntry
{
public:
    PhoneEntryI(const string& name, const string& phNum)
        : _name(name), _phNum(phNum)
    {
    }

    virtual string
    name(const Current&) override
    {
        return _name;
    }

    virtual string
    getNumber(const Current&) override
    {
        return _phNum;
    }

    virtual void
    setNumber(const string& phNum, const Current&) override
    {
        _phNum = phNum;
    }

    virtual void
    destroy(const Current& c) override
    {
        try
        {
           c.adapter->remove(c.id);
        }
        catch(const Ice::NotRegisteredException&)
        {
           throw Ice::ObjectNotExistException(__FILE__, __LINE__);
        }
    }

private:
    const string _name;
    string _phNum;
};
```

With this servant, `destroy` does just the right thing: it calls `delete` on the servant once the servant is idle, which in turn calls the destructor, so the memory used by the `_name` and `_phNum` data members is reclaimed.

However, real servants are rarely this simple. In particular, destruction of an Ice object may involve non-trivial actions, such as flushing a file,

committing a transaction, making a remote call on another object, or updating a hardware device. For example, instead of storing the details of a phone entry in member variables, the servant could be implemented to store the details in a file; in that case, destroying the Ice object would require closing the file. Seeing that the Ice run time calls the destructor of a servant only once the servant becomes idle, the destructor would appear to be an ideal place to perform such actions, for example:

C++11

```
class PhoneEntryI : public PhoneEntry
{
public:
    // ...

    ~PhoneEntryI()
    {
        _myStream.close(); // Bad idea
    }

private:
    fstream _myStream;
};
```

The problem with this code is that it can fail, for example, if the file system is full and buffered data cannot be written to the file. Such clean-up failure is a general issue for non-trivial servants: for example, a transaction can fail to commit, a remote call can fail if the network goes down, or a hardware device can be temporarily unresponsive.

If we encounter such a failure, we have a serious problem: we cannot inform the client of the error because, as far as the client is concerned, the `destroy` call completed just fine. The client will therefore assume that the Ice object was correctly destroyed. However, the system is now in an inconsistent state: the Ice object was destroyed (because its ASM entry was removed), but the object's state still exists (possibly with incorrect values), which can cause errors later.

Another reason for avoiding such state clean-up in C++ destructors is that destructors cannot throw exceptions: if they do, and do so in the process of being called during unwinding of the stack due to some other exception, the program goes directly to `terminate` and does not pass "Go". (There are a few exotic cases in which it is possible to throw from a destructor and get away with it but, in general, is an excellent idea to maintain the no-throw guarantee for destructors.) So, if anything goes wrong during destruction, we are in a tight spot: we are forced to swallow any exception that might be encountered by the destructor, and the best we can do is log the error, but not report it to the client.

Finally, using destructors to clean up servant state does not port well to languages such as Java and C#. For these languages, similar considerations apply to error reporting from a finalizer and, with Java, finalizers may not run at all. Therefore, we recommend that you perform any clean-up actions in the body of `destroy` instead of delaying clean-up until the servant's destructor runs.

Note that the foregoing does *not* mean that you cannot reclaim servant resources in destructors; after all, that is what destructors are for. But it *does* mean that you should not try to reclaim resources from a destructor if the attempt can fail (such as deleting records in an external system as opposed to, for example, deallocating memory or adjusting the value of variables in your program).

See Also

- Life Cycle and Normal Operations

Life Cycle and Collection Operations

On this page:

Factories and Collection Operations

The factory we defined earlier is what is known as a *pure* object factory because `create` is the only operation it provides. However, it is common for factories to do double duty and also act as collection managers that provide additional operations, such as `list` and `find`:

Slice

```
// ...

sequence<PhoneEntry*> PhoneEntries;

interface PhoneEntryFactory
{
    PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;

    idempotent PhoneEntry find(string name);
    idempotent PhoneEntries list();
}
```

`find` returns a proxy for the phone entry with the given name, and a null proxy if no such entry exists. `list` returns a sequence that contains the proxies of all existing entries.

Here is a simple implementation of `find`:

C++11

```
PhoneEntryPrx
PhoneEntryFactory::find(string name, const Current& c)
{
    auto comm = c.adapter->getCommunicator();

    shared_ptr<PhoneEntryPrx> pe;
    Identity id = Ice::stringToIdentity(name);
    if(c.adapter->find(id))
    {
        pe = Ice::uncheckedCast<PhoneEntryPrx>(c.adapter->createProxy(i
d));
    }
    return pe;
}
```

If an entry exists in the Active Servant Map (ASM) for the given name, the code creates a proxy for the corresponding Ice object and returns it. This code works correctly even for threaded servers: because the look-up of the identity in the ASM is atomic, there is no problem with other threads concurrently modifying the ASM (for example, while servicing calls from other clients to `create` or `destroy`).

Cyclic Dependencies between Factories and Objects

Unfortunately, implementing `list` is not as simple because it needs to iterate over the collection of entries, but the object adapter does not provide any iterator for ASM entries.

> The reason for this is that, during iteration, the ASM would have to be locked to protect it against concurrent access, but locking the ASM would prevent call dispatch during iteration and easily cause deadlocks.

Therefore, we must maintain our own list of entries inside the factory:

C++11

```
class PhoneEntryFactoryI : public PhoneEntryFactory
{
public:
    // ...

    void remove(const string& name, const shared_ptr<ObjectAdapter>&)
    {
        lock_guard<mutex> lock(_namesMutex);
        _names.erase(name);

    }

private:
    mutex _namesMutex;
    set<string> _names;
};
```

The idea is to have a set of names of existing entries, and to update that set in `create` and `destroy` as appropriate. However, for threaded servers, that raises a concurrency issue: if we have clients that can concurrently call `create`, `destroy`, and `list`, we need to interlock these operations to avoid corrupting the `_names` set (because STL containers are not thread-safe). This is the purpose of the mutex `_names Mutex` in the factory: `create`, `destroy`, and `list` can each lock this mutex to ensure exclusive access to the `_names` set.

Another issue is that our implementation of `destroy` must update the set of entries that is maintained by the factory. This is the purpose of the `remove` member function: it removes the specified name from the `_names` set (of course, under protection of the `_namesMutex` lock). However, `destroy` is a method on the `PhoneEntryI` servant, whereas `remove` is a method on the factory, so the servant must know how to reach the factory. Because the factory is a singleton, we can fix this by adding a static `_factory` member to the `PhoneEntryI` class:

C++11

```
class PhoneEntryI : public PhoneEntry
{
public:
    // ...

    static shared_ptr<PhoneEntryFactoryI> _factory;

private:
    const string _name;
    string _phNum;
};
```

The code in `main` then creates the factory and initializes the static member variable, for example:

C++11

```
PersonI::_factory = make_shared<PersonFactoryI>();

// Add factory to ASM and activate object
// adapter here...
```

This works, but it leaves a bad taste in our mouth because it sets up a cyclic dependency between the phone entry servants and the factory: the factory knows about the servants, and each servant knows about the factory so it can call `remove` on the factory. In general, such cyclic dependencies are a bad idea: if nothing else, they make a design harder to understand.

We could remove the cyclic dependency by moving the `_names` set and its associated mutex into a separate class instance that is referenced from both `PhoneEntryFactoryI` and `PhoneEntryI`. That would get rid of the cyclic dependency as far as the C++ type system is concerned but, as we will see later, it would not really help because the factory and its servants turn out to be mutually dependent regardless (because of concurrency issues). So, for the moment, we'll stay with this design and examine better alternatives after we have explored the concurrency issues in more detail.

With this design, we can implement `list` as follows:

C++11

```
PhoneEntries
PhoneEntryFactoryI::list(const Current& c)
{
    lock_guard<mutex> lock(_namesMutex);

    auto comm = c.adapter->getCommunicator();

    PhoneEntries pe;
    for(const auto& name : _names)
    {
        auto o = c.adapter->createProxy(Ice::stringToIdentity(name));
        pe.push_back(Ice::uncheckedCast<PhoneEntryPrx>(o));
    }

    return pe;
}
```

Note that `list` acquires a lock on the mutex, to prevent concurrent modification of the `_names` set by `create` and `destroy`. In turn, our `create` implementation now also locks the mutex:

C++11

```
shared_ptr<PhoneEntryPrx>
PhoneEntryFactory::create(string name, string phNum, const Current& c)
{
    lock_guard<mutex> lock(_namesMutex);

    PhoneEntryPrx pe;
    try
    {
        auto comm = c.adapter->getCommunicator();
        auto servant = make_shared<PhoneEntryI>(name, phNum);
        pe = Ice::uncheckedCast<PhoneEntryPrx>(c.adapter->add(servant,
Ice::stringToIdentity(name)));
    }
    catch (const Ice::AlreadyRegisteredException&)
    {
        throw PhoneEntryExists(name, phNum);
    }
    _names.insert(name);
    return pe;
}
```

With this implementation, we are safe if `create` and `list` run concurrently: only one of the two operations can acquire the lock at a time, so there is no danger of corrupting the `_names` set.

`destroy` is now trivial to implement: it simply removes the ASM entry and calls `remove` on the factory:

C++11

```
void
PhoneEntryI::destroy(const Current& c)
{
    // Note: not quite correct yet.
    c.adapter->remove(_name);
    _factory->remove(_name, c.adapter);
}
```

Note that this is not quite correct because we have not yet considered concurrency issues for destroy. We will consider this issue next.

See Also

- Object Creation
- The Active Servant Map

Life Cycle and Normal Operations

So far, we have mostly ignored the implementations of the `getNumber` and `setNumber` operations we defined in our discussion of object creation. Obviously, `getNumber` and `setNumber` must be interlocked against concurrent access — without this interlock, concurrent requests from clients could result in one thread writing to the `_phNum` member while another thread is reading it, with unpredictable results. (Conversely, the `name` operation need not have an interlock because the name of a phone entry is immutable.) To interlock `getNumber` and `setNumber`, we can add a mutex `_m` to `PhoneEntryI`:

C++11

```
class PhoneEntryI : public PhoneEntry
{
public:
    // ...

    static shared_ptr<PhoneEntryFactoryI> _factory;

private:
    const string _name;
    string _phNum;
    std::mutex _m;
};
```

The `getNumber` and `setNumber` implementations then lock `_m` to protect `_phNum` from concurrent access:

C++11

```
string
PhoneEntryI::name(const Current&)
{
    return _name;
}

string
PhoneEntryI::getNumber(const Current&)
{
    // Incorrect implementation!

    lock_guard<mutex> lock(_m);

    return _phNum;
}

void
PhoneEntryI::setNumber(const string& phNum, const Current&)
{
    // Incorrect implementation!

     lock_guard<mutex> lock(_m);

    _phNum = phNum;
}
```

This looks good but, as it turns out, `destroy` throws a spanner in the works: as shown, this code suffers from a rare, but real, race condition. Consider the situation where a client calls `destroy` at the same time as another client calls `setNumber`. In a server with a thread pool with more than one thread, the calls can be dispatched in separate threads and can therefore execute concurrently.

The following sequence of events can occur:

- The thread dispatching the `setNumber` call locates the servant, enters the operation implementation, and is suspended by the scheduler immediately on entry to the operation, before it can lock `_m`.
- The thread dispatching the `destroy` call locates the servant, enters `destroy`, successfully removes the servant from the Active Servant Map (ASM) and the `_names` set, and returns.
- The thread that was suspended in `setNumber` is scheduled again, locks `_m`, and now operates on a conceptually already-destroyed Ice object.

Note that, even if we lock `_m` in `destroy`, the problem persists:

C++11

```
void
PhoneEntryI::destroy(const Current& c)
{
    // Note: still not correct.

    lock_guard<mutex> lock(_m);

    c.adapter->remove(_name);
    _factory->remove(_name, c.adapter);
}
```

Even though `destroy` now locks `_m` and so cannot run concurrently with `getNumber` and `setNumber`, the preceding scenario can still arise. The problem here is that a thread can enter the servant and be suspended before it gets a chance to acquire a lock. With the code as it stands, this is not a problem: `setNumber` will simply update the `_phNum` member variable in a servant that no longer has an ASM entry. In other words, the Ice object is already destroyed — it just so happens that the servant for that Ice object is still hanging around because there is still an operation executing inside it. Any updates to the servant will succeed (even though they are useless because the servant's destruct or will run as soon as the last invocation leaves the servant.)

Note that this scenario is not unique to C++ and can arise even with Java synchronized operations: in that case, a thread can be suspended just after the Ice run time has identified the target servant, but before it actually calls the operation on the target servant. While the thread is suspended, another thread can execute `destroy`.

While this race condition does not affect our implementation, it does affect more complex applications, particularly if the servant modifies external state, such as a file system or database. For example, `setNumber` could modify a file in the file system; in that case, `destroy` woul d delete that file and probably close a file descriptor or stream. If we were to allow `setNumber` to continue executing after `destroy` has already done its job, we would likely encounter problems: `setNumber` might not find the file where it expects it to be or try to use the closed file descriptor and return an error; or worse, `setNumber` might end up re-creating the file in the process of updating the already-destroyed entry's phone number. (What exactly happens depends on how we write the code for each operation.)

Of course, we can try to anticipate these scenarios and handle the error conditions appropriately, but doing this for complex systems with complex servants rapidly gets out of hand: in each operation, we would have to ask ourselves what might happen if the servant is destroyed concurrently and, if so, take appropriate recovery action.

It is preferable to instead deal with interleaved invocations of `destroy` and other operations in a systematic fashion. We can do this by adding a `_destroyed` member to the `PhoneEntryI` servant. This member is initialized to false by the constructor and set to true by `destr oy`. On entry to every operation (including `destroy`), we lock the mutex, test the `_destroyed` flag, and throw `ObjectNotExistExceptio n` if the flag is set:

C++11

```
class PhoneEntryI : public PhoneEntry
{
public:
    // ...

    static shared_ptr<PhoneEntryFactoryI> _factory;

private:
    const string _name;
    string _phNum;
    bool   _destroyed;
    std::mutex _m;
};
```

```
PhoneEntryI::PhoneEntryI(const string& name, const string& phNum)
    : _name(name), _phNum(phNum), _destroyed(false)
{
}

string
PhoneEntryI::name(const Current&)
{
    lock_guard<mutex> lock(_m);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__);
    }

    return _name;
}

string
PhoneEntryI::getNumber(const Current&)
{
    lock_guard<mutex> lock(_m);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__);
    }
    return _phNum;
}

void
PhoneEntryI::setNumber(const string& phNum, const Current&)
{
    lock_guard<mutex> lock(_m);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__);
    }

    _phNum = phNum;
}

void
PhoneEntryI::destroy(const Current& c)
{
    lock_guard<mutex> lock(_m);

    if(_destroyed)
```

```
{
    throw ObjectNotExistException(__FILE__, __LINE__);
}

_destroyed = true;
c.adapter->remove(_name);
```

```
    _factory->remove(_name, c.adapter); // Dubious!
}
```

If you are concerned about the repeated code on entry to every operation, you can put that code into a member function or base class to make it reusable (although the benefits of doing so are probably too minor to make this worthwhile).

Using the `_destroyed` flag, if an operation is dispatched and suspended before it can lock the mutex and, meanwhile, `destroy` runs to completion in another thread, it becomes impossible for an operation to operate on the state of such a "zombie" servant: the test on entry to each operation ensures that any operation that runs after `destroy` immediately raises `ObjectNotExistException`.

Also note the "dubious" comment in `destroy`: the operation first locks `_m` and, while holding that lock, calls `remove` on the factory, which in turn locks its own `_namesMutex`. This is not wrong as such, but as we will see shortly, it can easily lead to deadlocks if we modify the application later.

See Also

- Object Creation
- The Active Servant Map
- Implementing a destroy Operation
- The Ice Threading Model

Removing Cyclic Dependencies

We mentioned earlier that factoring the `_names` set and its mutex into a separate class instance does not really solve the cyclic dependency problem, at least not in general. To see why, suppose that we want to extend our factory with a new `getDetails` operation:

Slice

```
// ...

struct Details
{
    PhoneEntry* proxy;
    string name;
    string phNum;
}

sequence<Details> DetailsSeq;

interface PhoneEntryFactory
{
    // ...

    DetailsSeq getDetails();
}
```

This type of operation is common in collection managers: instead of returning a simple list of proxies, `getDetails` returns a sequence of structures, each of which contains not only the object's proxy, but also some of the state of the corresponding object. The motivation for this is performance: with a plain list of proxies, the client, once it has obtained the list, is likely to immediately follow up with one or more remote calls for each object in the list in order to retrieve their state (for example, to display the list of objects to the user). Making all these additional remote procedure calls is inefficient, and an operation such as `getDetails` gets the job done with a single RPC instead.

To implement `getDetails` in the factory, we need to iterate over the set of entries and invoke the `getNumber` operation on each object. (These calls are collocated and therefore very efficient, so they do not suffer the performance problem that a client calling the same operations would suffer.) However, this is potentially dangerous because the following sequence of events is possible:

- Client A calls `getDetails`.
- The implementation of `getDetails` must lock `_namesMutex` to prevent concurrent modification of the `_names` set during iteration.
- Client B calls `destroy` on a phone entry.
- The implementation of `destroy` locks the entry's mutex `_m`, sets the `_destroyed` flag, and then calls `remove`, which attempts to lock `_namesMutex` in the factory. However, `_namesMutex` is already locked by `getDetails`, so `remove` blocks until `_m` is unlocked again.
- `getDetails`, while iterating over its set of entries, happens to call `getNumber` on the entry that is currently being destroyed by client B. `getNumber`, in turn, tries to lock its mutex `_m`, which is already locked by `destroy`.

At this point, the server deadlocks: `getDetails` holds a lock on `_namesMutex` and waits for `_m` to become available, and `destroy` holds a lock on `_m` and waits for `_namesMutex` to become available, so neither thread can make progress.

To get rid of the deadlock, we have two options:

- Rearrange the locking such that deadlock becomes impossible.
- Abandon the idea of calling back from the servants into the factory and use *reaping* instead.

We will explore both options in the following pages.

Topics

See Also

- Life Cycle and Collection Operations

Acquiring Locks without Deadlocks

For our example, it is fairly easy to avoid the deadlock caused by cyclic dependencies: instead of holding the lock for the duration of `destroy`, we set the `_destroyed` flag under protection of the lock and unlock `_m` again before calling `remove` on the factory:

C++11

```
void
PhoneEntryI::destroy(const Current& c)
{
    {
        lock_guard<mutex> lock(_m);

        if(_destroyed)
        {
            throw ObjectNotExistException(__FILE__, __LINE__);
        }

        _destroyed = true;

    } // _m is unlocked here.

    _factory->remove(_name, c.adapter);
}
```

Now deadlock is impossible because no function holds more than one lock, and no function calls another function while it holds a lock. However, rearranging locks in this fashion can be quite difficult for complex applications. In particular, if an application uses callbacks that do complex things involving several objects, it can be next to impossible to prove that the code is free of deadlocks. The same is true for applications that use condition variables and suspend threads until a condition becomes true.

At the core of the problem is that concurrency can create circular locking dependencies: an operation on the factory (such as `getDetails`) can require the same locks as a concurrent call to `destroy`. This is one reason why threaded code is harder to write than sequential code — the interactions among operations require locks, but dependencies among these locks are not obvious. In effect, locks set up an entirely separate and largely invisible set of dependencies. For example, it was easy to spot the mutual dependency between the factory and the servants due to the presence of `remove`; in contrast, it was much harder to spot the lurking deadlock in `destroy`. Worse, deadlocks may not be found during testing and discovered only after deployment, when it is much more expensive to rectify the problem.

See Also

- Removing Cyclic Dependencies

Reaping Objects

Instead of trying to arrange code such that it is deadlock-free in the presence of callbacks and cyclic dependencies, it is often easier to change the code to avoid the callbacks entirely and to use an approach known as *reaping*:

- `destroy` marks the servant as destroyed and removes the Active Servant Map (ASM) entry as usual, but it does not call back into the factory to update the `_names` set.
- Whenever a collection manager operation, such as `list`, `getDetails`, or `find` is called, the factory checks for destroyed servants and, if it finds any, removes them from the `_names` set.

Reaping can make for a much cleaner design because it avoids both the cyclic type dependency and the cyclic locking dependency.

On this page:

A Simple Reaping Implementation

To implement reaping, we need to change our `PhoneEntryI` definition a little. It no longer has a static `_factory` smart pointer back to the factory (because it no longer calls `remove`). Instead, the servant now provides a member function `_isZombie` that the factory calls to check whether the servant was destroyed some time in the past:

C++11

```
class PhoneEntryI : public PhoneEntry
{
public:
    // ...

    bool _isZombie() const;

private:
    const string _name;
    string _phNum;
    bool   _destroyed;
    std::mutex _m;
};
```

The implementation of `_isZombie` is trivial: it returns the `_destroyed` flag under protection of the lock:

C++11

```
bool
PhoneEntryI::_isZombie() const
{
    lock_guard<mutex> lock(_m);

    return _destroyed;
}
```

The `destroy` operation no longer calls back into the factory to update the `_names` set; instead, it simply sets the `_destroyed` flag and removes the ASM entry:

C++11

```
void
PhoneEntryI::destroy(const Current& c)
{
    lock_guard<mutex> lock(_m);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__);
    }

    _destroyed = true;
    c.adapter->remove(c.id);
}
```

The factory now, instead of storing just the names of existing servants, maintains a map that maps the name of each servant to its smart pointer:

C++11

```
class PhoneEntryFactoryI : public PhoneEntryFactory
{
public:
    // Constructor and Slice operations here...

private:
    using PMap = map<string, PhoneEntryIPtr>;
    PMap _entries;
    std::mutex _entriesMutex;
};
```

During `create` (and other operations, such as `list`, `getDetails`, and `find`), we scan for zombie servants and remove them from the `_entries` map:

C++11

```
shared_ptr<PhoneEntryPrx>
PhoneEntryFactory::create(string name, string phNum, const Current& c)
{
    lock_guard<mutex> lock(_entriesMutex);

    shared_ptr<PhoneEntryPrx> pe;
    auto servant = make_shared<PhoneEntryI>(name, phNum);

    // Try to create new object.
    //
    try
    {
        auto comm = c.adapter->getCommunicator();
        pe = Ice::uncheckedCast<PhoneEntryPrx>(
            c.adapter->add(servant, Ice::stringToIdentity(name)));
    }
    catch(const Ice::AlreadyRegisteredException&)
    {
        throw PhoneEntryExists(name, phNum);
    }

    // Scan for zombies.
    //
    auto i = _entries.begin();
    while(i != _entries.end())
    {
        if(i->second->_isZombie())
        {
            _entries.erase(i++);
        }
        else
        {
            ++i;
        }
    }
    _entries[name] = servant;

    return pe;
}
```

The implementations of `list`, `getDetails`, and `find` scan for zombies as well. Because they need to iterate over the existing entries anyway, reaping incurs essentially no extra cost:

C++11

```
PhoneEntries
PhoneEntryFactoryI::list(const Current& c)
{
    lock_guard<mutex> lock(_entriesMutex);

    auto comm = c.adapter->getCommunicator();

    PhoneEntries pe;
    for(auto i = _entries.begin(); i != _entries.end();)
    {
        if(i->second->_isZombie())
        {
            _entries.erase(i++);
        }
        else
        {
            auto
o = c.adapter->createProxy(Ice::stringToIdentity(i->first));
            pe.push_back(Ice::uncheckedCast<PhoneEntryPrx>(o));
            ++i;
        }
    }

    return pe;
}

// Similar for getDetails and find...
```

This is a much cleaner design: there is no cyclic dependency between the factory and the servants, either implicit (in the type system) or explicit (as a locking dependency). Moreover, the implementation is easier to understand once you get used to the idea of reaping: there is no need to follow complex callbacks and to carefully analyze the order of lock acquisition. (Note that, depending on how state is maintained for servants, you may also need to reap during start-up and shutdown.)

In general, we recommend that you use a reaping approach in preference to callbacks for all but the most trivial applications: it simply is a better approach that is easier to maintain and understand.

Alternative Reaping Implementations

You may be concerned that reaping increases the cost of `create` from *O*(log *n*) to *O*(*n*) because `create` now iterates over all existing entries and locks and unlocks a mutex in each servant (whereas, previously, it simply added each new servant to the `_names` set). Often, this is not an issue because life cycle operations are called infrequently compared to normal operations. However, you will notice the additional cost if you have a large number of servants (in the thousands or more) and life cycle operations are called frequently.

If you find that `create` is a bottleneck (by profiling, not by guessing!), you can change to a more efficient implementation by adding zombie servants to a separate zombie list. Reaping then iterates over the zombie list and removes each servant in the zombie list from the `_entries` map before clearing the zombie list. This reduces the cost of reaping to be proportional to the number of zombie servants instead of the total number of servants. In addition, it allows us to remove the `_isZombie` member function and to lock and unlock `_entriesMutex` only once instead of locking a mutex in each servant as part of `_isZombie`. We will see such an implementation when we revisit the file system application.

You may also be concerned about the number of zombie servants that can accumulate in the server if `create` is not called for some time. For most applications, this is not a problem: the servants occupy memory, but no other resources because `destroy` can clean up scarce

resources, such as file descriptors or network connections before it turns the servant into a zombie. If you really need to prevent accumulation of zombie servants, you can reap from a background thread that runs periodically, or you can count the number of zombies and trigger a reaping pass once that number exceeds some threshold.

See Also

- Removing Cyclic Dependencies
- The Active Servant Map
- Object Life Cycle for the File System Application

Object Identity and Uniqueness

When creating an object with a factory, it may not be a good idea to allow clients to control the object identity of the Ice objects they create. Here is the scenario from our previous discussion, re-cast in terms of our phone book application:

1. Client A creates a new phone entry for Fred.
2. Client A passes the proxy for the Fred entry as a parameter of a remote call to another part of the system, say, server B.
3. Server B remembers Fred's proxy.
4. Client A decides that the entry for Fred is no longer needed and calls Fred's destroy operation.
5. Some time later, client C creates a new phone entry for a different person whose name also happens to be Fred.
6. Server B decides to get Fred's phone number by calling `getNumber` on the proxy it originally obtained from client A.

At this point, things are likely to go wrong: server B thinks that it has obtained the phone number of the original Fred, but that entry no longer exists and has since been replaced by a new entry for a different person (who presumably has a different phone number).

What has happened here is that Fred has been reincarnated because the same object identity was used for two different objects. In general, such reused object identities are a bad idea. For example, consider the following interfaces:

Slice

```
interface Process
{
    void launch(); // Start process.
}

interface Missile
{
    void launch(); // Kill lots of people.
}
```

Replaying the preceding scenario, if client A creates a `Process` object called "Thunderbird" and destroys that object again, and client C creates a `Missile` object called "Thunderbird", when server B calls `launch`, it will launch a missile instead of a process.

To be fair, in reality, this scenario is unlikely because it tacitly assumes that both objects are implemented by the same object adapter but, in a realistic scenario, the same server would be unlikely to implement both launching of processes and missiles. However, if you have objects that derive from common base interfaces, so objects of different types share the same operation names, this problem is real: operation invocations can easily end up in a destroyed and later recreated object.

Specifically, the preceding scenario illustrates that, when the Ice run time dispatches a request, exactly three items determine where the request ends up being processed:

- the endpoint at which the server listens for incoming requests
- the identity of the Ice object that is the target of the request
- the name of the operation that is to be invoked on the Ice object

If object identities are insufficiently unique, a request intended for one object can end up being sent to a completely different object, provided that the original object used the same identity, that both provide an operation with the same name, and that the parameters passed to one operation happen to decode correctly when interpreted as the parameters to the other operation. (This is rare, but not impossible, depending on the type and number of parameters.)

The crucial question is, what do we mean by "insufficiently unique"? As far as the call dispatch is concerned, identities must be unique only per object adapter. This is because the ASM does not allow you to add two entries with the same object identity; by enforcing this, the Active Servant Map (ASM) ensures that each object identity belongs to exactly one servant. (Note that the converse, namely, that servants in ASM entries must be unique, is *not* the case: the ASM allows you to map different object identities to the same servant, which is useful to, for example, implement stateless facade objects — default servants are used for this purpose.) So, as far as the Ice run time is concerned, it is perfectly OK to reuse object identities for different Ice objects.

Note that the Ice run time cannot prevent reuse of object identities either. Doing so would require the run time to remember every object identity that has ever been used, which does not scale. Instead, the Ice run time makes the application responsible for ensuring that object identities are "sufficiently unique".

You can deal with the identity reuse problem in several ways. One option is to do nothing and simply ignore the problem. While this sounds facetious, it is a viable option for many applications because, due to their nature, identity reuse is simply impossible. For example, if you use a social security number as a person's identity, the problem cannot arise because the social security number of a deceased person is not

given to another person.

Another option is to allow identity reuse and to write your application such that it can deal with such identities: if nothing bad happens when an identity is reused, there is no problem. (This is the case if you know that the life cycles of the proxies for two different objects with the same identity can never overlap.)

The third option is to ensure that object identities are guaranteed unique, for example, by establishing naming conventions that make reuse impossible, or by using a UUID as the object identity. This can be useful even for applications for which identity reuse does not pose a problem. For example, if you use well-known objects, globally-unique object identities allow you to move a server to a different machine without invalidating proxies to these objects that are held by clients.

In general, we recommend that if an Ice object naturally contains a unique item of state (such as a social security number), you should use that item as the object identity. On the other hand, if the natural object identity is insufficiently unique (as is the case with names of phone book entries), you should use a UUID as the identity. (This is particularly useful for anonymous transient objects, such as session objects, that may not have a natural identity.)

See Also

- Object Creation
- Object Identity
- The Active Servant Map
- Default Servants
- Servant Activation and Deactivation
- Understanding Object Life Cycle
- Well-Known Objects

Object Life Cycle for the File System Application

Now that we have had a look at the issues around object life cycle, let us return to our file system application and add life cycle operations to it, so clients can create and destroy files and directories.

To destroy a file or directory, the obvious choice is to add a `destroy` operation to the `Node` interface:

Slice

```
module Filesystem
{
    exception GenericError
    {
        string reason;
    }

    exception PermissionDenied extends GenericError {}
    exception NameInUse extends GenericError {}
    exception NoSuchName extends GenericError {}

    interface Node
    {
        idempotent string name();
        void destroy() throws PermissionDenied;
    }

    // ...
}
```

Note that `destroy` can throw a `PermissionDenied` exception. This is necessary because we must prevent attempts to destroy the root directory.

The `File` interface is unchanged:

Slice

```
module Filesystem
{
    // ...

    sequence<string> Lines;

    interface File extends Node
    {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    }

    // ...
}
```

Note that, because `File` derives from `Node`, it inherits the `destroy` operation we defined for `Node`.

The `Directory` interface now looks somewhat different from the previous version:

- The `list` operation returns a sequence of structures instead of a list of proxies: for each entry in a directory, the `NodeDesc` structure provides the name, type, and proxy of the corresponding file or directory.
- Directories provide a `find` operation that returns the description of the nominated node. If the nominated node does not exist, the operation throws a `NoSuchName` exception.
- The `createFile` and `createDirectory` operations create a file and directory, respectively. If a file or directory already exists, the operations throw a `NameInUse` exception.

Here are the corresponding definitions:

Slice

```
module Filesystem
{
    // ...

    enum NodeType { DirType, FileType }

    struct NodeDesc
    {
        string name;
        NodeType type;
        Node* proxy;
    }

    sequence<NodeDesc> NodeDescSeq;

    interface Directory extends Node
    {
        idempotent NodeDescSeq list();
        idempotent NodeDesc find(string name) throws NoSuchName;
        File* createFile(string name) throws NameInUse;
        Directory* createDirectory(string name) throws NameInUse;
    }
};
```

Note that this design is somewhat different from the factory we designed for the phone book application. In particular, we do not have a single object factory; instead, we have as many factories as there are directories, that is, each directory creates files and directories only in that directory.

The motivation for this design is twofold:

- Because all files and directories that can be created are immediate descendants of their parent directory, we avoid the complexities of parsing path names for a separator such as "/". This keeps our example code to manageable size. (A real-world implementation of a distributed file system would, of course, be able to deal with path names.)
- Having more than one object factory presents interesting implementation issues that we will explore in the following discussion.

Let's move on to the implementation of this design in C++ and Java. You can find the full code of the implementation (including languages other than C++ and Java) in the `Manual/lifecycle` directory of your Ice distribution.

Topics

See Also

- Slice for a Simple File System
- Object Creation

Implementing Object Life Cycle in C++

The implementation of our life cycle design has the following characteristics:

- It uses UUIDs as the object identities for nodes to avoid object reincarnation problems.
- When `destroy` is called on a node, the node needs to destroy itself and inform its parent directory that it has been destroyed (because the parent directory is the node's factory and also acts as a collection manager for child nodes).

Note that, in contrast to the initial version, the entire implementation resides in a `FilesystemI` namespace instead of being part of the `Filesystem` namespace. Doing this is not essential, but is a little cleaner because it keeps the implementation in a namespace that is separate from the Slice-generated namespace.

On this page:

Object Life Cycle Changes for the `NodeI` Class in C++

To begin with, let us look at the definition of the `NodeI` class:

C++11

```
namespace FilesystemI
{
    class DirectoryI;

    class NodeI : public virtual Filesystem::Node
    {
    public:

        virtual std::string name(const Ice::Current&) override;
        Ice::Identity id() const;

    protected:

        NodeI(const std::string&, const std::shared_ptr<DirectoryI>&);

        const std::string _name;
        const std::shared_ptr<DirectoryI> _parent;
        bool _destroyed;
        Ice::Identity _id;
        std::mutex _mutex;
    };
...
}
```

The purpose of the `NodeI` class is to provide the data and implementation that are common to both `FileI` and `DirectoryI`, which use implementation inheritance from `NodeI`.

As in the initial version, `NodeI` provides the implementation of the `name` operation and stores the name of the node and its parent directory in the `_name` and `_parent` members. (The root directory's `_parent` member is null.) These members are immutable and initialized by the constructor and, therefore, `const`.

The `_destroyed` member, protected by the mutex `_m`, prevents the race condition we discussed earlier. The constructor initializes `_destroyed` to `false` and creates an identity for the node (stored in the `_id` member):

C++11

```
FilesystemI::NodeI::NodeI(const string& nm, const
shared_ptr<DirectoryI>& parent)
    : _name(nm), _parent(parent), _destroyed(false)
{
    //
    // Create an identity. The root directory has the fixed identity
"RootDir".
    //
    if(parent != nullptr)
    {
        _id.name = Ice::generateUUID();
    }
    else
    {
        _id.name = "RootDir";
    }
}
```

The `id` member function returns a node's identity, stored in the `_id` data member. The node must remember this identity because it is a UUID and is needed when we create a proxy to the node:

C++11

```
Identity
FilesystemI::NodeI::id() const
{
    return _id;
}
```

The data members of `NodeI` are protected instead of private to keep them accessible to the derived `FileI` and `DirectoryI` classes. (Because the implementation of `NodeI` and its derived classes is quite tightly coupled, there is little point in making these members private and providing separate accessors and mutators for them.)

The implementation of the Slice `name` operation simply returns the name of the node, but also checks whether the node has been destroyed:

C++11

```
string
FilesystemI::NodeI::name(const Current& c)
{
    lock_guard<mutex> lock(_mutex);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet,
c.operation);
    }

    return _name;
}
```

This completes the implementation of the `NodeI` base class.

Object Life Cycle Changes for the `DirectoryI` Class in C++

Next, we need to look at the implementation of directories. The `DirectoryI` class derives from `NodeI` and the Slice-generated `Directory` skeleton class. Of course, it must implement the pure virtual member functions for its Slice operations, which leads to the following (not yet complete) definition:

C++11

```
namespace FilesystemI
{
    class DirectoryI : public NodeI, public Filesystem::Directory,
public std::enable_shared_from_this<DirectoryI>
    {
    public:

        virtual Filesystem::NodeDescSeq list(const Ice::Current&)
override;
        virtual Filesystem::NodeDesc find(std::string, const
Ice::Current&) override;
        std::shared_ptr<Filesystem::FilePrx> createFile(std::string,
const Ice::Current&) override;
        std::shared_ptr<Filesystem::DirectoryPrx>
createDirectory(std::string, const Ice::Current&) override;
        virtual void destroy(const Ice::Current&) override;

        // ...
    };
}
```

Each directory stores its contents in a map that maps the name of a directory to its servant:

C++11

```
namespace FilesystemI
{
    class DirectoryI : public NodeI, public Filesystem::Directory,
public std::enable_shared_from_this<DirectoryI>
    {
    public:
        // ...

        DirectoryI(const std::string& = "/", const
std::shared_ptr<DirectoryI>& = nullptr);

        void removeEntry(const std::string&);

    private:

        using Contents = std::map<std::string, std::shared_ptr<NodeI>>;
        Contents _contents;
    };
}
```

Note that we use the inherited member `_m` to interlock operations.

The constructor simply initializes the `NodeI` base class:

C++11

```
FilesystemI::DirectoryI::DirectoryI(const string& name,
const shared_ptr<DirectoryI>& parent)
    : NodeI(name, parent)
{
}
```

The `removeEntry` member function is called by the child to remove itself from its parent's `_contents` map:

C++11

```
void
FilesystemI::DirectoryI::removeEntry(const string& name)
{
    lock_guard<mutex> lock(_m);
    _contents.erase(name);
}
```

Here is the `destroy` member function for directories:

C++11

```
void
FilesystemI::DirectoryI::destroy(const Current& c)
{
    if(!_parent)
    {
        throw PermissionDenied("Cannot destroy root directory");
    }
    else
    {
        lock_guard<mutex> lock(_mutex);

        if(_destroyed)
        {
            throw ObjectNotExistException(__FILE__, __LINE__, c.id,
c.facet, c.operation);
        }

        if(!_contents.empty())
        {
            throw PermissionDenied("Cannot destroy non-empty
directory");
        }

        c.adapter->remove(id());
        _destroyed = true;
    }

    _parent->removeEntry(_name);
}
```

The code first prevents destruction of the root directory and then checks whether this directory was destroyed previously. It then acquires the lock and checks that the directory is empty. Finally, `destroy` removes the Active Servant Map (ASM) entry for the destroyed directory and removes itself from its parent's `_contents` map. Note that we call `removeEntry` outside the synchronization to avoid deadlocks.

The `createDirectory` implementation locks the mutex before checking whether the directory already contains a node with the given name (or an invalid empty name). If not, it creates a new servant, adds it to the ASM and the `_contents` map, and returns its proxy:

C++11

```
shared_ptr<DirectoryPrx>
FilesystemI::DirectoryI::createDirectory(string nm, const Current& c)
{
    lock_guard<mutex> lock(_mutex);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet,
c.operation);
    }

    if(nm.empty() || _contents.find(nm) != _contents.end())
    {
        throw NameInUse(nm);
    }

    auto d = make_shared<DirectoryI>(nm, shared_from_this());
    auto node = c.adapter->add(d, d->id());
    _contents[nm] = d;
    return Ice::uncheckedCast<DirectoryPrx>(node);
}
```

The `createFile` implementation is identical, except that it creates a file instead of a directory:

C++11

```
shared_ptr<FilePrx>
FilesystemI::DirectoryI::createFile(string nm, const Current& c)
{
    lock_guard<mutex> lock(_mutex);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet,
c.operation);
    }

    if(nm.empty() || _contents.find(nm) != _contents.end())
    {
        throw NameInUse(nm);
    }

    auto f = make_shared<FileI>(nm, shared_from_this());
    auto node = c.adapter->add(f, f->id());
    _contents[nm] = f;
    return Ice::uncheckedCast<FilePrx>(node);
}
```

Here is the implementation of `list`:

C++11

```
NodeDescSeq
FilesystemI::DirectoryI::list(const Current& c)
{
    lock_guard<mutex> lock(_mutex);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet,
c.operation);
    }

    NodeDescSeq ret;
    for(const auto& i: _contents)
    {
        NodeDesc d;
        d.name = i.first;
        d.type = dynamic_pointer_cast<File>(i.second) ?
NodeType::FileType : NodeType::DirType;
        d.proxy =
Ice::uncheckedCast<NodePrx>(c.adapter->createProxy(i.second->id()));
        ret.push_back(d);
    }
    return ret;
}
```

After acquiring the lock, the code iterates over the directory's contents and adds a `NodeDesc` structure for each entry to the returned vector.

The `find` operation proceeds along similar lines:

C++11

```
NodeDesc
FilesystemI::DirectoryI::find(string nm, const Current& c)
{
    lock_guard<mutex> lock(_mutex);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet,
c.operation);
    }

    auto pos = _contents.find(nm);
    if(pos == _contents.end())
    {
        throw NoSuchName(nm);
    }

    auto p = pos->second;
    NodeDesc d;
    d.name = nm;
    d.type = dynamic_pointer_cast<File>(p) ? NodeType::FileType :
NodeType::DirType;
    d.proxy =
Ice::uncheckedCast<NodePrx>(c.adapter->createProxy(p->id()));
    return d;
}
```

Object Life Cycle Changes for the `FileI` Class in C++

The constructor of `FileI` is trivial: it simply initializes the data members of its base class::

C++11

```
FilesystemI::FileI::FileI(const string& nm, const
shared_ptr<DirectoryI>& parent)
    : NodeI(nm, parent)
{
}
```

The implementation of the three member functions of the `FileI` class is also trivial, so we present all three member functions here:

C++11

```
Lines
FilesystemI::FileI::read(const Current& c)
{
    lock_guard<mutex> lock(_mutex);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet,
c.operation);
    }

    return _lines;
}

void
FilesystemI::FileI::write(Lines text, const Current& c)
{
    lock_guard<mutex> lock(_mutex);

    if(_destroyed)
    {
        throw ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet,
c.operation);
    }

    _lines = move(text);
}

void
FilesystemI::FileI::destroy(const Current& c)
{
    {
        lock_guard<mutex> lock(_mutex);

        if(_destroyed)
        {
            throw ObjectNotExistException(__FILE__, __LINE__, c.id,
c.facet, c.operation);
        }

        c.adapter->remove(id());
        _destroyed = true;
    }

    _parent->removeEntry(_name);
}
```

Object Life Cycle Concurrency Issues in C++

The preceding implementation is provably deadlock free. All member functions hold only one lock at a time, so they cannot deadlock with each other or themselves. While the locks are held, the functions do not call other member functions that acquire locks, so any potential deadlock can only arise by concurrent calls to another mutating function, either on the same node or on different nodes. For concurrent calls on the same node, deadlock is impossible because such calls are strictly serialized on the mutex `_m`; for concurrent calls to `destroy` on different nodes, each node locks its respective mutex `_m`, releases `_m` again, and then acquires and releases a lock on its parent (by calling `removeEntry`), also making deadlock impossible.

See Also

- Example of a File System Server in C++98
- Life Cycle and Normal Operations
- Acquiring Locks without Deadlocks
- Object Identity and Uniqueness
- The Active Servant Map

Implementing Object Life Cycle in Java

The implementation of our life cycle design has the following characteristics:

- It uses UUIDs as the object identities for nodes to avoid object reincarnation problems.
- When `destroy` is called on a node, the node needs to destroy itself and inform its parent directory that it has been destroyed (because the parent directory is the node's factory and also acts as a collection manager for child nodes).

Note that, in contrast to the initial version, the entire implementation resides in a `FilesystemI` package instead of being part of the `Filesystem` package. Doing this is not essential, but is a little cleaner because it keeps the implementation in a package that is separate from the Slice-generated package.

On this page:

Object Life Cycle Changes for the `NodeI` Class in Java

Our `DirectoryI` and `FileI` servants derive from a common `NodeI` base interface. This interface is not essential, but useful because it allows us to treat servants of type `DirectoryI` and `FileI` polymorphically:

Java

```
package FilesystemI;

public interface NodeI
{
    com.zeroc.Ice.Identity id();
}
```

The only method is the `id` method, which returns the identity of the corresponding node.

Object Life Cycle Changes for the `DirectoryI` Class in Java

As in the initial version, the `DirectoryI` class derives from the generated base class `_DirectoryDisp`. In addition, the class implements the `NodeI` interface. `DirectoryI` must implement each of the Slice operations, leading to the following outline:

Java

```
package FilesystemI;

import com.zeroc.Ice.*;
import Filesystem.*;

public class DirectoryI implements Directory, NodeI
{
    public Identity id();

    public synchronized String name(Current c);

    public synchronized NodeDesc[] list(Current c);

    public synchronized NodeDesc find(String name, Current c)
        throws NoSuchName;

    public synchronized FilePrx createFile(String name, Current c)
        throws NameInUse;

    public synchronized DirectoryPrx createDirectory(String name, Curre
nt c)
        throws NameInUse;

    public void destroy(Current c)
        throws PermissionDenied;

    // ...
}
```

To support the implementation, we also require a number of methods and data members:

Java

```
package FilesystemI;

import com.zeroc.Ice.*;
import Filesystem.*;

public class DirectoryI implements Directory, NodeI
{
    // ...

    public DirectoryI();
    public DirectoryI(String name, DirectoryI parent);

    public synchronized void removeEntry(String name);

    private String _name;       // Immutable
    private DirectoryI _parent; // Immutable
    private Identity _id;       // Immutable
    private boolean _destroyed;
    private java.util.Map<String, NodeI> _contents;
}
```

The `_name` and `_parent` members store the name of this node and a reference to the node's parent directory. (The root directory's `_parent` member is null.) Similarly, the `_id` member stores the identity of this directory. The `_name`, `_parent`, and `_id` members are immutable once they have been initialized by the constructor. The `_destroyed` member prevents a race condition; to interlock access to `_destroyed` (as well as the `_contents` member) we can use synchronized methods (as for the `name` method), or use a `synchronized(this)` block.

The `_contents` map records the contents of a directory: it stores the name of an entry, together with a reference to the child node.

Here are the two constructors for the class:

Java

```
public DirectoryI()
{
    this("/", null);
}

public DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;
    _id = new Identity();
    _destroyed = false;
    _contents = new java.util.HashMap<>();

    _id.name = parent == null ? "RootDir" : java.util.UUID.randomUUID()
.toString();
}
```

The first constructor is a convenience function to create the root directory with the fixed identity "RootDir" and a null parent.

The real constructor initializes the `_name`, `_parent`, `_id`, `_destroyed`, and `_contents` members. Note that nodes other than the root directory use a UUID as the object identity.

The `removeEntry` method is called by the child to remove itself from its parent's `_contents` map:

Java

```
public synchronized void removeEntry(String name)
{
    _contents.remove(name);
}
```

The implementation of the Slice name operation simply returns the name of the node, but also checks whether the node has been destroyed:

Java

```
public synchronized String name(Current c)
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }
    return _name;
}
```

Note that this method is synchronized, so the `_destroyed` member cannot be accessed concurrently.

Here is the `destroy` member function for directories:

Java

```
public void destroy(Current c)
    throws PermissionDenied
{
    if(_parent == null)
    {
        throw new PermissionDenied("Cannot destroy root directory");
    }

    synchronized(this)
    {
        if(_destroyed)
        {
            throw new ObjectNotExistException();
        }

        if(_contents.size() != 0)
        {
            throw new PermissionDenied("Cannot destroy non-empty direct
ory");
        }

        c.adapter.remove(id());
        _destroyed = true;
    }

    _parent.removeEntry(_name);
}
```

The code first prevents destruction of the root directory and then checks whether this directory was destroyed previously. It then acquires the lock and checks that the directory is empty. Finally, `destroy` removes the Active Servant Map (ASM) entry for the destroyed directory and removes itself from its parent's `_contents` map. Note that we call `removeEntry` outside the synchronization to avoid deadlocks.

The `createDirectory` implementation acquires the lock before checking whether the directory already contains a node with the given name (or an invalid empty name). If not, it creates a new servant, adds it to the ASM and the `_contents` map, and returns its proxy:

Java

```
public synchronized DirectoryPrx createDirectory(String name, Current c)
    throws NameInUse
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }

    if(name.length() == 0 || _contents.containsKey(name))
    {
        throw new NameInUse(name);
    }

    DirectoryI d = new DirectoryI(name, this);
    ObjectPrx node = c.adapter.add(d, d.id());
    _contents.put(name, d);
    return DirectoryPrx.uncheckedCast(node);
}
```

The `createFile` implementation is identical, except that it creates a file instead of a directory:

Java

```
public synchronized FilePrx createFile(String name, Current c)
    throws NameInUse
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }

    if(name.length() == 0 || _contents.containsKey(name))
    {
        throw new NameInUse(name);
    }

    FileI f = new FileI(name, this);
    ObjectPrx node = c.adapter.add(f, f.id());
    _contents.put(name, f);
    return FilePrx.uncheckedCast(node);
}
```

Here is the implementation of `list`:

Java

```
public synchronized NodeDesc[] list(Current c)
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }
    NodeDesc[] ret = new NodeDesc[_contents.size()];
    java.util.Iterator<java.util.Map.Entry<String, NodeI> > pos =
_contents.entrySet().iterator();
    for(int i = 0; i < _contents.size(); ++i)
    {
        java.util.Map.Entry<String, NodeI> e = pos.next();
        NodeI p = e.getValue();
        ret[i] = new NodeDesc();
        ret[i].name = e.getKey();
        ret[i].type = p instanceof FileI
? NodeType.FileType : NodeType.DirType;
        ret[i].proxy = NodePrx.uncheckedCast(c.adapter.createProxy(p.id
()));
    }
    return ret;
}
```

After acquiring the lock, the code iterates over the directory's contents and adds a `NodeDesc` structure for each entry to the returned array.

The `find` operation proceeds along similar lines:

Java

```
public synchronized NodeDesc find(String name, Current c)
    throws NoSuchName
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }

    NodeI p = _contents.get(name);
    if(p == null)
    {
        throw new NoSuchName(name);
    }

    NodeDesc d = new NodeDesc();
    d.name = name;
    d.type = p instanceof FileI ? NodeType.FileType : NodeType.DirType;
    d.proxy = NodePrx.uncheckedCast(c.adapter.createProxy(p.id()));
    return d;
}
```

Object Life Cycle Changes for the `FileI` Class in Java

The `FileI` class is similar to the `DirectoryI` class. The data members store the name, parent, and identity of the file, as well as the `_destroyed` flag and the contents of the file (in the `_lines` member). The constructor initializes these members:

Java

```
package FilesystemI;

import com.zeroc.Ice.*;
import Filesystem.*;
import FilesystemI.*;

public class FileI implements File, NodeI
{
    // ...

    public FileI(String name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;
        _destroyed = false;
        _id = new Identity();
        _id.name = java.util.UUID.randomUUID().toString();
    }

    private String _name;
    private DirectoryI _parent;
    private boolean _destroyed;
    private Identity _id;
    private String[] _lines;
}
```

The implementation of the remaining member functions of the `FileI` class is trivial, so we present all of them here:

Java

```
public synchronized String name(Current c)
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }
    return _name;
}

public Identity id()
{
    return _id;
}

public synchronized String[] read(Current c)
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }

    return _lines;
}

public synchronized void write(String[] text, Current c)
{
    if(_destroyed)
    {
        throw new ObjectNotExistException();
    }

    _lines = (String[])text.clone();
}

public void destroy(Current c)
{
    synchronized(this)
    {
        if(_destroyed)
        {
            throw new ObjectNotExistException();
        }

        c.adapter.remove(id());
        _destroyed = true;
    }

    _parent.removeEntry(_name);
}
```

Object Life Cycle Concurrency Issues in Java

The preceding implementation is provably deadlock free. All methods hold only one lock at a time, so they cannot deadlock with each other or themselves. While the locks are held, the methods do not call other methods that acquire locks, so any potential deadlock can only arise by concurrent calls to another mutating method, either on the same node or on different nodes. For concurrent calls on the same node, deadlock is impossible because such calls are strictly serialized on the instance; for concurrent calls to `destroy` on different nodes, each node locks itself, releases itself again, and then acquires and releases a lock on its parent (by calling `removeEntry`), also making deadlock impossible.

See Also

- Example of a File System Server in Java
- Life Cycle and Normal Operations
- Object Identity and Uniqueness
- Acquiring Locks without Deadlocks
- The Active Servant Map

Platform-Specific Features

This section describes Ice features that are available only on a specific platform, such as Microsoft Windows.

Topics

Windows Services

A Windows service is a program that runs in the background and typically does not require user intervention. Similar to a daemon on Unix platforms, a Windows service is usually launched automatically when the operating system starts and runs until the system shuts down.

Ice includes the `Service` class that simplifies the task of writing an Ice-based Windows service in C++. Writing the service is only the first step, however, as it is also critically important that the service be installed and configured correctly for successful operation. Service installation and configuration is outside the scope of the `Service` class because these tasks are generally not performed by the service itself but rather as part of a larger administrative effort to deploy an application. Furthermore, there are security implications to consider when a service is able to install itself: such a service typically needs administrative rights to perform the installation, but does not necessarily need those rights while it is running. A better strategy is to grant administrative rights to a separate installer program and not to the service itself.

Topics

Installing a Windows Service

The installation of a Windows service varies in complexity with the needs of the application, but usually involves the following activities:

- Selecting the user account in which the service will run.
- Registering the service and establishing its activation mode and dependencies.
- Creating one or more file system directories to contain executables, libraries, and supporting files or databases.
- Configuring those directories with appropriate permissions so that they are accessible to the user account selected for the service.
- Creating keys in the Windows registry.
- Configuring the Windows Event Log so that the service can report status and error messages.

There are many ways to perform these tasks. For example, an administrator can execute them manually. Another option is to write a script or program tailored to the needs of your application. Finally, you can build an installer using a developer tool such as InstallShield.

Selecting a User Account for the Service

Before installing a service, you should give careful consideration to the user account that will run the service. Unless your service has special requirements, we recommend that you use the built-in account that Windows provides specifically for this purpose, `Local Service`.

See Also

- Manually Installing a Service as a Windows Service

Using the Ice Service Installer

Ice provides the command-line tool `iceserviceinstall` to assist you in installing and uninstalling the following Ice services as Windows services:

- IceGrid registry
- IceGrid node
- Glacier2 router

Ice includes other programs that can also be run as Windows services, such as the IceBox and IcePatch2 servers. Typically it is not necessary to install these programs as Windows services because they can be launched by an IceGrid node service. However, if you wish to run an IceBox or IcePatch2 server as a Windows service without the use of IceGrid, you must manually install the service.

Here we describe how to use the Ice service installer and discuss its actions and prerequisites.

On this page:

`iceserviceinstall` Command Line Options

`iceserviceinstall` supports the following options and arguments:

```
iceserviceinstall [options] service config-file [property ...]

Options:
-h, --help           Show this message.
-n, --nopause        Do not call pause after displaying a message.
-v, --version        Display the Ice version.
-u, --uninstall      Uninstall the Windows service.
```

The *`service`* and *`config-file`* arguments are required during installation and uninstallation.

The *`service`* argument selects the type of service you are installing; use one of the following values:

- `icegridregistry`
- `icegridnode`
- `glacier2router`

Note that the Ice service installer currently does not support the installation of an IceGrid node with a collocated registry, therefore you must install the registry and node separately.

The *`config-file`* argument specifies the name of an Ice configuration file.

When installing a service, properties can be defined on the command line using the *`--name=value`* syntax, or they can be defined in the configuration file. The supported properties are described below.

Security Considerations for Ice Services

None of the Ice services require privileges beyond a normal user account. In the case of the IceGrid node service in particular, we do not recommend running it in a user account with elevated privileges because the service is responsible for launching server executables, and those servers would inherit the node's access rights.

`iceserviceinstall` Configuration File

The Ice service installer requires that you specify the path name of the Ice configuration file for the service being installed or uninstalled. The tool needs this path name for several reasons:

- During installation, it verifies that the configuration file has sufficient access rights.
- It configures a newly-installed service to load the configuration file using its absolute path name, therefore you must decide in advance where the file will be located.
- It reads the configuration file and examines certain service-specific properties. For example, prior to installing an IceGrid registry service, the tool verifies that the directory specified by the property `IceGrid.Registry.Data` has sufficient access rights.
- The tool supports its own configuration parameters that may also be defined as properties in this file.

You may still modify a service's configuration file after installation, but you should uninstall and reinstall the service if you change any of the properties that influence the service installer's actions. The table below describes the service properties that affect the installer:

Property	Service	Description
`IceGrid.InstanceName`	IceGrid Registry	Value appears in the service name; also included in the default display name if one is not defined.
`IceGrid.Node.Data`	IceGrid Node	Directory is created if necessary; access rights are verified.
`IceGrid.Node.Name`	IceGrid Node	Value appears in the service name; also included in the default display name if one is not defined.
`IceGrid.Registry.Data`	IceGrid Registry	Directory is created if necessary; access rights are verified.
`Ice.Default.Locator`	IceGrid Node, Glacier2 Router	The IceGrid instance name is derived from the identity in this proxy.
`Ice.EventLog.Source`	All	Specifies the name of an event log source for the service.

The steps performed by the tool during an installation are described in detail below.

Sample Configuration Files

Ice includes sample configuration files for the IceGrid and Glacier2 services in the `config` subdirectory of your Ice installation. We recommend that you review the comments and settings in these files to familiarize yourself with a typical configuration of each service.

You can modify a configuration file to suit your needs or copy one to use as a starting point for your own configuration.

`iceserviceinstall` Properties

The Ice service installer uses a set of optional properties that customize the installation process. These properties can be defined in the service's configuration file as discussed above, or they can be defined on the command line using the familiar *--name=value* syntax:

```
iceserviceinstall --DependOnRegistry=1 ...
```

The installer's properties are listed below:

- AutoStart=*num*

Num value	Service Startup Type
0	Manual
1	Automatic
2	Automatic (Delayed Start)

  If not specified, the default *num* value is 1. You should select 2, Automatic (Delayed Start), when your service is listening on a

Wireless LAN interface.

- Debug=*num*
  If *num* is a value greater than zero, iceserviceinstall outputs diagnostics when installing and uninstalling a server. If not specified, the default value is 0.
- `DependOnRegistry=`*num*
  If num is a value greater than zero, the service is configured to depend on the IceGrid registry, meaning Windows will start the registry prior to starting this service. Enabling this feature also requires that the property `Ice.Default.Locator` be defined in *config-file*. If not specified, the default value is zero.
- `Description=`*value*
  A brief description of the service. If not specified, a general description is used.
- `DisplayName=`*name*
  The friendly name that identifies the service to the user. If not specified, `iceserviceinstall` composes a default display name.
- `EventLog=`*name*
  The name of the event log used by the service. If not specified, the default value is `Application`.
- `ImagePath=`*path*
  The path name of the service executable. If not specified, `iceserviceinstall` assumes the service executable resides in the same directory as itself and fails if the executable is not found.
- `ObjectName=`*name*
  Specifies the account used to run the service. If not specified, the default value is `NT Authority\LocalService`.
- `Password=`*value*
  The password required by the account specified in `ObjectName`.

Service Installation Process

The Ice service installer performs a number of steps to install a service. As discussed earlier, you must specify the path name of the service's configuration file because the service installer uses certain properties during the installation process. The actions taken by the service installer are described below:

- Obtain the service's *instance name* from the configuration file. The instance name is specified by the property `IceGrid.InstanceName` or `Glacier2.InstanceName`. If an instance name is not specified, the default value is `IceGrid` or `Glacier2`, respectively. If the service being installed depends on the IceGrid registry, the IceGrid instance name is derived from the value of the `Ice.Default.Locator` property.
- For an IceGrid node, obtain the node's name from the property `IceGrid.Node.Name`. This property must be defined when installing a node.
- Compose the service name from the service type, instance name, and node name (for an IceGrid node). For example, the default service name for an IceGrid registry is `icegridregistry.IceGrid`. Note that the service name is not the same as the display name.
- Resolve the user account specified by `ObjectName`.
- Grant `ObjectName` read and execute permissions on the parent directory of `ImagePath`.
- For an IceGrid registry, create the data directory specified by the property `IceGrid.Registry.Data` and ensure that the user account specified by `ObjectName` has read/write access to the directory.
- For an IceGrid node, create the data directory specified by the property `IceGrid.Node.Data` and ensure that the user account specified by `ObjectName` has read/write access to the directory.
- For an IceGrid node, ensure that the user account specified by `ObjectName` has read access to the following registry key:
  `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib`
  This key allows the node to access CPU utilization statistics.
- Ensure that the user account specified by `ObjectName` has read access to the configuration file.
- Create a new Windows event log by adding the registry key specified by `EventLog`.
- Add an event log source under `EventLog` for the source name specified by `Ice.EventLog.Source`. If this property is not defined, the service name is used as the source name.
- Install the service, including command line arguments that specify the service name (`--service`*name*) and the absolute path name of the configuration file (`--Ice.Config=`*config-file*).

The Ice service installer currently does not verify that the user account specified by `ObjectName` has the right to "Log on as a service".

Uninstalling a Windows Service

When uninstalling an existing service, the Ice service installer first ensures that the service is stopped, then proceeds to remove the service. The service's event log source is removed and, if the service is not using the `Application` log, the event log registry key is also removed.

See Also

- icegridregistry
- icegridnode
- Getting Started with Glacier2
- IceBox
- IcePatch2
- Installing a Windows Service
- Manually Installing a Service as a Windows Service
- Troubleshooting Windows Services
- IceGrid.*
- Glacier2.*

Manually Installing a Service as a Windows Service

This page describes how to manually install and configure an Ice service as a Windows Service using the IceBridge service as a case study. For the purposes of this discussion, we assume that Ice is installed in the directory `C:\Program Files\ZeroC\Ice-3.7.0`. We also assume that you have administrative access to your system, which is required by many of the installation steps discussed below.

On this page:

Selecting a User Account for the IceBridge Service

The IceBridge service can run in a regular user account, therefore we will follow our own recommendation and use the Local Service account.

Configuration File for the IceBridge Service

IceBridge requires a minimal set of configuration properties. We could specify them on the service's command line, but if we later want to modify those properties we would have to reinstall the service. Defining the properties in a file simplifies the task of modifying the service's configuration.

Our sample IceBridge configuration is quite simple:

```
IceBridge.Source.Endpoints=tcp -p 10000
IceBridge.Target.Endpoints=tcp -h localhost -p 21112
Ice.Trace.Network=2
```

We will save our configuration properties into the following file:

```
C:\ProgramData\Ice\icebridge.cfg
```

We must also ensure that the service has permission to access its configuration file. The Ice run time never modifies a configuration file, therefore read access is sufficient. The configuration file likely already has the necessary access rights, which we can verify using the `icacls` utility:

```
> icacls C:\ProgramData\Ice\icebridge.cfg
```

Creating the IceBridge Service

We will use Microsoft's Service Control (`sc`) utility in a command window to create the service.

> See http://support.microsoft.com/kb/251192 for more information about the `SC` utility.

Our first `sc` command does the majority of the work (the command is formatted for readability but must be typed on a single line):

```
> sc create icebridge binPath= "C:\Program
Files\ZeroC\Ice-3.7.0\bin\icebridge.exe
--Ice.Config=C:\ProgramData\Ice\icebridge.cfg --service icebridge"
  DisplayName= "IceBridge Server" start= auto obj= "NT
Authority\LocalService" password= ""
```

There are several important aspects of this command:

- You must execute this command in an Administrator command prompt.
- The service name is `icebridge`. You can use whatever name you like, as long as it does not conflict with an existing service. Note however that this name is used in other contexts, such as in the `--service` option discussed below, therefore you must use it consistently.
- Following the service are several options. Note that all of the option names end with an equals sign and are separated from their arguments with at least one space.
- The `binPath=` option is required. We supply the full path name of the IceBridge server executable, as well as command-line arguments that define the location of the configuration file and the name of the service, all enclosed in quotes.
- The `DisplayName=` option sets a friendly name for the service.
- The `start=` option configures the start up behavior for the service. We used the argument `auto` to indicate the service should be started automatically when Windows boots.
- The `obj=` option selects the user account in which this service runs. As we explained, the `Local Service` account is appropriate for most services.
- The `password=` option supplies the password associated with the user account indicated by `obj=`. The `Local Service` account has an empty password.

The `sc` utility should report success if it was able to create the service as specified. You can verify that the new service was created with this command:

```
> sc qc icebridge
```

Alternatively, you can start the Services administrative control panel and inspect the properties of the IceBridge service.

If you start the control panel, you will notice that the entry for IceBridge does not have a description. To add a description for the service, use the following command:

```
> sc description icebridge "Simple IceBridge Server"
```

After refreshing the list of services, you should see the new description.

Creating the Event Log for the IceBridge Service

By default, programs such as the IceBridge service that utilize the `Service` class log messages to the `Application` event log. Below we describe how to prepare the Windows registry for the service's default behavior, and we also show how to use a custom event log instead. We make use of Microsoft's Registry (`reg`) utility to modify the registry, although you could also use the interactive `regedit` tool. As always, caution is recommended whenever you modify the registry.

Using the Application Log for the IceBridge Service

We must configure an event log source for events to display properly. The first step is to create a registry key with the name of the source. Since the `Service` class uses the service name as the source name by default, we add the key `icebridge` as shown below:

```
> reg add
HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\icebridge
```

Inside this key we must add a value specifies the location of the Ice run time DLL:

```
> reg add
HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\icebridge /v
EventMessageFile /t REG_EXPAND_SZ /d "C:\Program
Files\ZeroC\Ice-3.7.0\bin\ice37.dll"
```

We will also add a value indicating the types of events that the source supports:

```
> reg add
HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\icebridge /v
TypesSupported /t REG_DWORD /d 7
```

The value 7 corresponds to the combination of the following event types:

- `EVENTLOG_ERROR_TYPE`
- `EVENTLOG_WARNING_TYPE`
- `EVENTLOG_INFORMATION_TYPE`

You can verify that the registry values have been defined correctly using the following command:

```
> reg query
HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\icebridge
```

Our configuration of the event log is now complete.

Changing the Source Name for the IceBridge Service

Using the configuration described in the previous section, events logged by the IceBridge service are recorded in the event log using the source name `icebridge`. If you prefer to use a source name that differs from the service name, you can replace `icebridge` in the registry commands with the name of your choosing, but you must also add a matching definition for the property `Ice.EventLog.Source` to the service's configuration file.

For example, to use the source name `Ice Bridging Service`, you would add the registry key as shown below:

```
> reg add
"HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\Ice
Bridging Service"
```

The commands to add the `EventMessageFile` and `TypesSupported` values must be modified in a similar fashion. Finally, add the following configuration property to `icebridge.cfg`:

```
Ice.EventLog.Source=Ice Bridging Service
```

Using a Custom Log for the IceBridge Service

You may decide that you want your services to record messages into an application-specific log instead of the `Application` log that is shared by other unrelated services. As an example, let us create a log named `MyApp`:

```
> reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\MyApp"
```

Next we add a subkey for the IceBridge service. As described in the previous section, we will use a friendlier source name:

```
> reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\MyApp\Ice
Bridging Service"
```

Now we can define values for `EventMessageFile` and `TypesSupported`:

```
> reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\MyApp\Ice
Bridging Service" /v EventMessageFile /t REG_EXPAND_SZ /d "C:\Program
Files\ZeroC\Ice-3.6.0\bin\ice37.dll"

> reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\MyApp\Ice
Bridging Service" /v TypesSupported /t REG_DWORD /d 7
```

Finally, we define `Ice.EventLog.Source` in the IceBridge service's configuration file:

```
Ice.EventLog.Source=Ice Bridging Service
```

Note that you must restart the Event Viewer control panel after adding the `MyApp` registry key in order to see the new log.

Registry Caching for the IceBridge Service

The first time a service logs an event, Windows' Event Log service caches the registry entries associated with the service's source. If you wish to modify a service's event log configuration, such as changing the service to use a custom log instead of the `Application` log, you should perform the following steps:

1. Stop the service.
2. Remove the unwanted event log registry key.
3. Add the new event log registry key(s).
4. Restart the system (or at least the Windows Event Log service).
5. Start the service and verify that the log entries appear in the intended log.

After following these steps, open a log entry and ensure that it displays properly. If it does not, for example if the event properties indicate that the description of an event cannot be found, the problem is likely due to a misconfigured event source. Verify that the value of `EventMessageFile` refers to the correct location of the Ice run time DLL, and that the service is defining `Ice.EventLog.Source` in its configuration file (if necessary).

Starting the IceBridge Service

We are at last ready to start the service. In a command window, you can use the `sc` utility:

```
> sc start icebridge
```

The program usually responds with status information indicating that the start request is pending. You can query the service's status to verify that it started successfully:

```
> sc query icebridge
```

The service should now be in the running state. If it is not in this state, open the Event Viewer control panel and inspect the relevant log for more information that should help you to locate the problem. Even if the service started successfully, you may still want to use the Event Viewer to confirm that the service is using the log you expected.

See Also

- Service Helper Class
- Installing a Windows Service

Troubleshooting Windows Services

This page describes how to troubleshoot Windows Services.

On this page:

Missing Libraries for a Windows Service

One failure that commonly occurs when starting a Windows service is caused by missing DLLs, which usually results in an error window stating a particular DLL cannot be found. Fixing this problem can often be a trial-and-error process because the DLL mentioned in the error may depend on other DLLs that are also missing. It is important to understand that a Windows service is launched by the operating system and can be configured to execute as a different user, which means the service's environment (most importantly its `PATH`) may not match yours and therefore extra steps are necessary to ensure that the service can locate its required DLLs.

> The command-line utility `dumpbin` can be used to discover the dependencies of an executable or DLL.

The simplest approach is to copy all of the necessary DLLs to the directory containing the service executable. If this solution is undesirable, another option is to modify the system `PATH` to include the directory or directories containing the required DLLs. (Note that modifying the system `PATH` requires restarting the system.) Finally, you can copy the necessary DLLs to `\WINDOWS\system32`, although we do not recommend this approach.

> Copying DLLs to `\WINDOWS\system32` often results in subtle problems later when trying to develop using newer versions of the DLLs. Inevitably you will forget about the DLLs in `\WINDOWS\system32` and struggle to determine why your application is misbehaving or failing to start.

Assuming that DLL issues are resolved, a Windows service can fail to start for a number of other reasons, including

- invalid command-line arguments or configuration properties
- inability to access necessary resources such as file systems and databases, because either the resources do not exist or the service does not have sufficient access rights to them
- networking issues, such as attempting to open a port that is already in use, or DNS lookup failures

Failures encountered by the Ice run time prior to initialization of the communicator are reported to the Windows event log if no other logger implementation is defined, so that should be the first place you look. Typically you will find an entry in the `System` event log resembling the following message:

```
The IceBridge service terminated with service-specific error 1.
```

Error code `1` corresponds to `EXIT_FAILURE`, the value used by the `Service` class to indicate a failure during startup. Additional diagnostic messages may be available in the `Application` event log. See `Ice::Service` Logging Considerations for more information on configuring a logger for a Windows service.

As we mentioned earlier, insufficient access rights can also prevent a Windows service from starting successfully. By default, a Windows service is configured to run under a local system account, in which case the service may not be able to access resources owned by other users. It may be necessary for you to configure a service to run under a different account, which you can do using the Services control panel. You should also review the access rights of files and directories required by the service.

Windows Firewall Interference

Your choice of user account determines whether you receive any notification when the Windows Firewall blocks the ports that are used by your service. For example, if you use `Local Service` as we recommended, you will not see any Windows Security Alert dialog (see this Microsoft article for details).

If you are not prompted to unblock your service, you will need to manually add an exception in Windows Firewall. For example, follow the steps below to unblock the ports of a Glacier2 router service:

1. Open the Windows Firewall Settings panel and navigate to the Exceptions panel.
2. Select "Add program..."
3. Select "Browse," navigate to the Glacier2 router executable, and click "OK."

Note that adding an exception for a program unblocks all ports on which the program listens. Review the endpoint configurations of your services carefully to ensure that no unnecessary ports are opened.

For services listening on one or a few fixed ports, you could also create port exceptions in your Windows Firewall. Refer to the Windows Firewall documentation for details.

IceGrid Node Performance Monitoring Issues

The IceGrid node uses Windows' `Perflib` facility to obtain statistics about the CPU utilization of its host for load balancing purposes. Occasionally, the IceGrid node may log the following warning message:

```
warning: Unable to lookup the performance counter name
```

This message is an indication that the node does not have sufficient privileges to access a key in the Windows registry:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib
```

As part of its installation procedure, the `iceserviceinstall` utility modifies the permissions of this registry key to grant read access to the node's designated user account. If you are trying to change the node's user account, we recommend using the `iceserviceinstall` utility to uninstall and reinstall the node. If you wish to modify the permissions of this registry key manually, follow these steps:

1. Start `regedit` and navigate to the `Perflib` key.
2. Right click on `Perflib` and select `Permissions`.
3. If the desired user account is not already present, click `Add` to add the user account. Enter `LOCAL SERVICE` if you wish to run the node in the Local Service account, otherwise enter the name of the user account. Press `OK`.
4. Check the `Read` box in the `Allow` column to grant read access to the registry key and press `OK` to apply the changes.

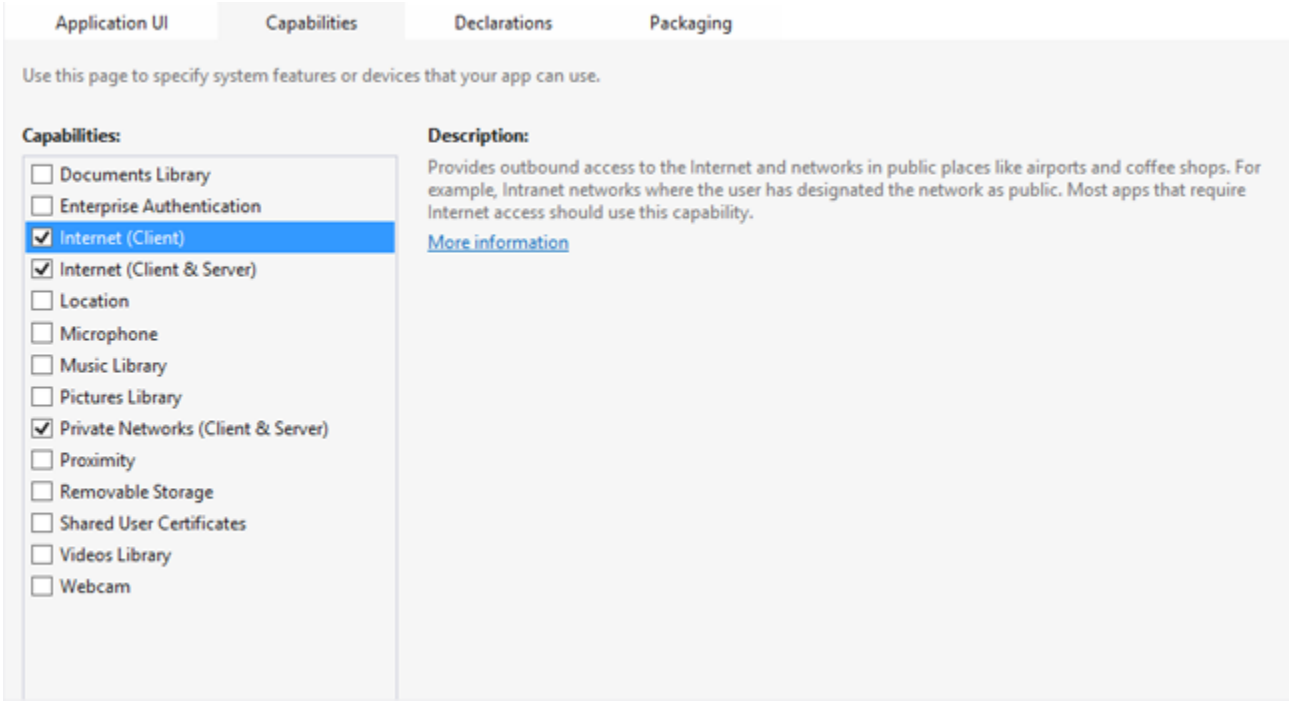
Another way to grant the node's user account with the necessary access rights is to add it to the `Performance Monitor Users` group.

See Also

- Load Balancing
- Installing a Windows Service
- Manually Installing a Service as a Windows Service

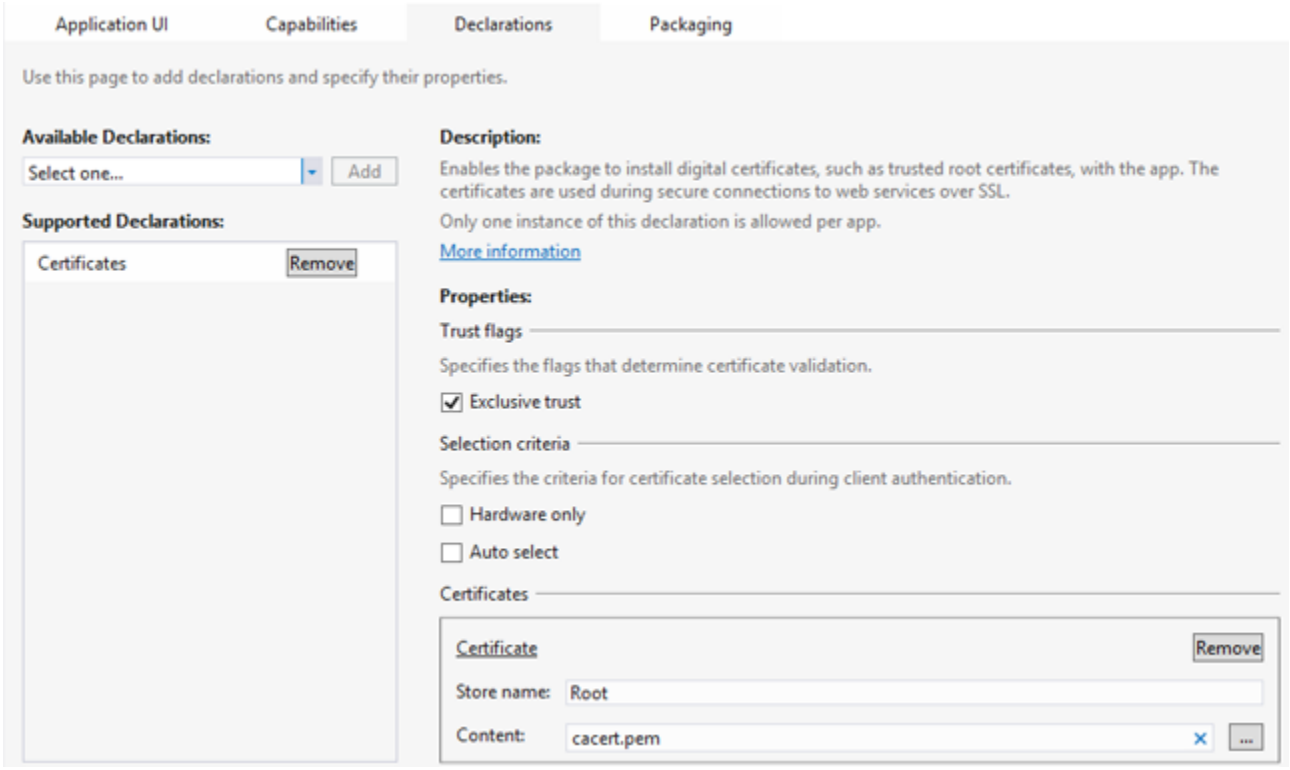
Windows Store Applications

A Windows Store application that uses Ice must request certain capabilities in its manifest in order to access public and private networks. The image below shows the relevant settings in Visual Studio:



Neglecting to enable the proper capabilities can raise `Ice::SocketException` or, if you are debugging the application in Visual Studio at the time of the exception, it will likely break on `Platform::AccessDeniedException`.

If your client application uses SSL, you can bundle certificates with the application by including them in the application's declarations:



Example
The UWP sample programs in the Ice distribution use capabilities and declarations.

Property Reference

This section provides a reference for all properties used by the Ice run time and its services.

Unless stated otherwise in the description of an individual property, its default value is the empty string. If a property takes a numeric value, the empty string is interpreted as zero.

Note that Ice reads properties that control the run time and its services only once on start-up, when you create a communicator. This means that you must set Ice-related properties to their correct values before you create a communicator. If you change the value of an Ice-related property after that point, it is likely that the new setting will simply be ignored.

Topics

Object Adapter Properties

On this page:

adapter.ACM.Close

Synopsis

`adapter.ACM.Close=num`

Description

Overrides the value of `Ice.ACM.Server.Close` for incoming connections to the named object adapter.

Note that ACM can cause incoming oneway requests to be silently discarded.

adapter.ACM.Heartbeat

Synopsis

`adapter.ACM.Heartbeat=num`

Description

Overrides the value of `Ice.ACM.Server.Heartbeat` for incoming connections to the named object adapter.

adapter.ACM.Timeout

Synopsis

`adapter.ACM.Timeout=num`

Description

Overrides the value of `Ice.ACM.Server.Timeout` for incoming connections to the named object adapter.

adapter.AdapterId

Synopsis

adapter.AdapterId=*id*

Description

Specifies an identifier for the object adapter with the name *adapter*. This identifier must be unique among all object adapters using the same locator instance. If a locator proxy is defined using *adapter*.Locator or Ice.Default.Locator, this object adapter sets its endpoints with the locator registry upon activation.

adapter.Endpoints

Synopsis

adapter.Endpoints=*endpoints*

Description

Sets the endpoints for the object adapter *adapter* to *endpoints*. These endpoints specify the network interfaces on which the object adapter receives requests. Proxies created by the object adapter contain these endpoints, unless the *adapter*.PublishedEndpoints pro perty is also specified.

adapter.Locator

Synopsis

adapter.Locator=*locator*

Description

Specifies a locator for the object adapter with the name *adapter*. The value is a stringified proxy to the Ice locator interface.

As a proxy property, you can configure additional aspects of the proxy using properties.

adapter.MessageSizeMax

Synopsis

adapter.MessageSizeMax=*num*

Description

Overrides the setting of Ice.MessageSizeMax to limit the size of messages that can be received by this object adapter. If not defined, the adapter uses the value of Ice.MessageSizeMax.

adapter.ProxyOptions

Synopsis

adapter.ProxyOptions=*options*

Description

Specifies the proxy options for proxies created by the object adapter. The value is a string representing the proxy options as they would be specified in a stringified proxy. The default value is "-t", that is, proxies created by the object adapter are configured to use twoway invocations by default.

adapter.PublishedEndpoints

Synopsis

`adapter.PublishedEndpoints=endpoints`

Description

When creating a proxy, the object adapter *adapter* normally includes the endpoints defined by *adapter*`.Endpoints`. If *adapter*`.PublishedEndpoints` is defined, the object adapter publishes these endpoints instead. This is useful in many situations, such as when a server resides behind a port-forwarding firewall, in which case the object adapter's public endpoints must specify the address and port of the firewall. The *adapter*`.ProxyOptions` property also influences the proxies created by an object adapter.

adapter.ReplicaGroupId

Synopsis

`adapter.ReplicaGroupId=id`

Description

Identifies the group of replicated object adapters to which this adapter belongs. The replica group is treated as a virtual object adapter, so that an indirect proxy of the form `identity@`*id* refers to the object adapters in the group. During binding, a client will attempt to establish a connection to an endpoint of one of the participating object adapters, and automatically try others until a connection is successfully established or all attempts have failed. Similarly, an outstanding request will, when permitted, automatically fail over to another object adapter of the replica group upon connection failure. The set of endpoints actually used by the client during binding is determined by the locator's configuration policies.

Defining a value for this property has no effect unless *adapter*`.AdapterId` is also defined. Furthermore, the locator registry may require replica groups to be defined in advance (see `IceGrid.Registry.DynamicRegistration`), otherwise `Ice.NotRegisteredException` is raised upon adapter activation. Regardless of whether an object adapter is replicated, it can always be addressed individually in an indirect proxy if it defines a value for *adapter*`.AdapterId`.

adapter.Router

Synopsis

`adapter.Router=router`

Description

Specifies a router for the object adapter with the name *adapter*. The value is a stringified proxy to the Ice router control interface. Defining a router allows the object adapter to receive callbacks from the router over a bidirectional connection, thereby avoiding the need for the router to establish a connection back to the object adapter.

A router can only be assigned to one object adapter. Specifying the same router for more than one object adapter results in undefined behavior. The default value is no router.

As a proxy property, you can configure additional aspects of the proxy using properties.

adapter.ThreadPool.Serialize

Synopsis

`adapter.ThreadPool.Serialize=num`

Description

If *num* is a value greater than 0, the adapter's thread pool serializes all messages from each connection. It is not necessary to enable this feature in a thread pool whose maximum size is 1 thread. When a thread pool dispatches requests implemented with AMD, it serializes the dispatching of requests from each connection, but it does not wait for a request to complete before it dispatches the next request.

In a multi-threaded pool, enabling serialization allows requests from different connections to be dispatched concurrently while preserving the order of messages on each connection. Note that serialization can have a significant impact on latency and throughput. If not defined, the default value is 0.

adapter.ThreadPool.Size

Synopsis

`adapter.ThreadPool.Size=num`

Description

A communicator creates a default server thread pool that dispatches requests to its object adapters. An object adapter can also be configured with its own thread pool. This is useful in avoiding deadlocks due to thread starvation by ensuring that a minimum number of threads is available for dispatching requests to certain Ice objects.

num is the initial number of threads in the thread pool. The default value is 0, meaning that an object adapter by default uses the communicator's server thread pool. See `Ice.ThreadPool.name.Size` for more information.

adapter.ThreadPool.SizeMax

Synopsis

`adapter.ThreadPool.SizeMax=num`

Description

num is the maximum number of threads for the thread pool. See `Ice.ThreadPool.name.SizeMax` for more information.

The default value is the value of `adapter.ThreadPool.Size`, meaning the thread pool can never grow larger than its initial size.

adapter.ThreadPool.SizeWarn

Synopsis

`adapter.ThreadPool.SizeWarn=num`

Description

Whenever *num* threads are active in a thread pool, a "low on threads" warning is printed. The default value is 0, which disables the warning.

adapter.ThreadPool.StackSize

Synopsis

`adapter.ThreadPool.StackSize=num`

Description

num is the stack size (in bytes) of threads in the thread pool. The default value is 0, meaning the operating system's default is used.

adapter.ThreadPool.ThreadIdleTime

Synopsis

```
adapter.ThreadPool.ThreadIdleTime=num
```

Description

In a dynamically-sized thread pool, Ice reaps a thread after it is idle for *num* seconds. Setting this property to 0 disables idle thread reaping. If not specified, the default value is 60 seconds. See `Ice.ThreadPool.name.ThreadIdleTime` for more information.

adapter.ThreadPool.ThreadPriority

Synopsis

```
adapter.ThreadPool.ThreadPriority=num
```

Description

num specifes a thread priority for the object adapter's thread pool. The object adapter creates its threads with the specified priority. Leaving this property unset causes the adapter to create threads with the priority specified by `Ice.ThreadPool.Server.ThreadPriority` or, if that property is unset, the priority specified by `Ice.ThreadPriority`.

Proxy Properties

The communicator operation `propertyToProxy` creates a proxy from a group of configuration properties. The argument to `propertyToProxy` is a string representing the base name of the property group (shown as *name* in the property descriptions below). This name must correspond to a property that supplies the stringified form of the proxy. Subordinate properties can be defined to customize the proxy's local configuration.

The communicator operation `proxyToProperty` performs the inverse operation, that is, returns the property group for a proxy.

On this page:

name

Synopsis

name=*proxy*

Description

The base property of the group with an application-specific *name* supplying the stringified representation of a proxy. Use the communicator operation `propertyToProxy` to retrieve the property and convert it into a proxy.

name.CollocationOptimized

Synopsis

name.CollocationOptimized=*num*

Description

If *num* is a value greater than zero, the proxy is configured to use collocated invocations when possible. Defining this property is equivalent to invoking the `ice_collocationOptimized` proxy method.

name.ConnectionCached

Synopsis

name.ConnectionCached=*num*

Description

If *num* is a value greater than zero, the proxy caches its chosen connection for use in subsequent requests. Defining this property is equivalent to invoking the `ice_connectionCached` proxy method.

name.Context.*key*

Synopsis

name.Context.*key*=*value*

Description

Adds the key/value pair to the proxy's request context.

name.EndpointSelection

Synopsis

name.EndpointSelection=*type*

Description

Specifies the proxy's endpoint selection type. Legal values are `Random` and `Ordered`. Defining this property is equivalent to invoking the `ice_endpointSelection` proxy method.

name.InvocationTimeout

Synopsis

name.InvocationTimeout=*num*

Description

Specifies the invocation timeout to be used by this proxy. Defining this property is equivalent to invoking the `ice_invocationTimeout` proxy method.

name.Locator

Synopsis

name.Locator=*proxy*

Description

Specifies the proxy of the locator to be used by this proxy. Defining this property is equivalent to invoking the `ice_locator` proxy method.

This is a proxy property, so you can configure additional local aspects of the proxy with subordinate properties. For example:

```
MyProxy.Locator=...
MyProxy.Locator.EndpointSelection=Ordered
```

name.LocatorCacheTimeout

Synopsis

name.LocatorCacheTimeout=*num*

Description

Specifies the locator cache timeout to be used by this proxy. Defining this property is equivalent to invoking the `ice_locatorCacheTimeout` proxy method.

name.PreferSecure

Synopsis

name`.PreferSecure=`*num*

Description

If *num* is a value greater than zero, the proxy gives precedence to secure endpoints. If not defined, the proxy uses the value of `Ice.Default.PreferSecure`.

Defining this property is equivalent to invoking the `ice_preferSecure` proxy method.

name.Router

Synopsis

name`.Router=`*proxy*

Description

Specifies the proxy of the router to be used by this proxy. Defining this property is equivalent to invoking the `ice_router` proxy method.

This is a proxy property, so you can configure additional local aspects of the proxy with subordinate properties. For example:

```
MyProxy.Router=...
MyProxy.Router.PreferSecure=1
```

Miscellaneous Ice.* Properties

On this page:

Ice.BackgroundLocatorCacheUpdates

Synopsis

`Ice.BackgroundLocatorCacheUpdates=`*`num`*

Description

If *`num`* is set to 0 (the default), an invocation on an indirect proxy whose endpoints are older than the configured locator cache timeout triggers a locator cache update; the run time delays the invocation until the new endpoints are returned by the locator.

If *`num`* is set to a value larger than 0, an invocation on an indirect proxy with expired endpoints still triggers a locator cache update, but the update is performed in the background, and the run time uses the expired endpoints for the invocation. This avoids delaying the first invocation that follows expiry of a cache entry.

Ice.BatchAutoFlushSize

Synopsis

`Ice.BatchAutoFlushSize=`*`num`*

Description

This property controls how the Ice run time deals with flushing of batch messages. If *num* is set to a value greater than 0, the run time automatically forces a flush of the current batch when a new message is added to a batch and that message would cause the batch to exceed *num* kilobytes. If *num* is set to 0 or a negative number, batches must be flushed explicitly by the application. If not defined, the default value is `1024`.

> When flushed, batch requests are sent as a single Ice message. The Ice run time in the receiver limits incoming messages to the maximum size specified by `Ice.MessageSizeMax`, therefore the sender must periodically flush batch requests (whether manually or automatically) to ensure they do not exceed the receiver's configured limit.

Ice.CacheMessageBuffers

Synopsis

`Ice.CacheMessageBuffers=`*num* (Java, .NET)

Description

If *num* is a value greater than 0, the Ice run time caches message buffers for future reuse. This can improve performance and reduce the amount of garbage produced by Ice internals that the garbage collector would eventually spend time to reclaim. However, for applications that exchange very large messages, this cache may consume excessive amounts of memory and therefore should be disabled by setting this property to 0.

> This property affects the caching of message buffers for synchronous invocation, asynchronous invocation, and synchronous dispatch. The Ice run time never caches message buffers for asynchronous dispatch.

Platform Notes

- Java
  Ice allocates non-direct message buffers when this property is set to 1 and direct message buffers when set to 2. Use of direct message buffers minimizes copying and typically results in improved throughput. If not defined, the default value is 2.
- .NET
  If not defined, the default value is 1.

Ice.ChangeUser

Synopsis

`Ice.ChangeUser=`*user* (C++ & Unix only)

Description

If set, Ice changes the user and group id to the respective ids of *user* in `/etc/passwd`. This only works if the Ice application is executed by the super-user.

Ice.ClassGraphDepthMax

Synopsis

`Ice.ClassGraphDepthMax=`*num*

Description

Specifies the maximum depth for a graph of Slice class instances to unmarshal. If this maximum is reached, the Ice run time throws an `Ice::MarshalException`. Reading and destroying a Slice class graph are recursive operations. This property prevents stack overflows from occurring if a sender sends a very large graph and not enough space on the stack is available. To read larger graphs, you can increase the value of this property but you should also make sure the stack size is still large enough or increase it using the thread pool StackSize propert y. If not specified, the default value is 100.

If this is not a concern, you can set `Ice.ClassGraphDepthMax` to 0; setting this property to 0 (or to a negative number) disables the depth limit altogether.

Ice.CollectObjects

Synopsis

`Ice.CollectObjects=`*`num`* (C++98)

Description

Ice for C++ includes a garbage collection facility for reclaiming cyclic graphs of Slice class instances that are unmarshaled by the Ice run time. Setting this property to `1` causes the Ice run time to assume that all cyclic object graphs that a program receives are eligible for collection by default. If not specified, the default value is 0.

Ice.Compression.Level

Synopsis

`Ice.Compression.Level=`*`num`*

Description

Specifies the bzip2 compression level to use when compressing protocol messages. Legal values for *`num`* are `1` to `9`, where `1` represents the fastest compression and `9` represents the best compression. Note that higher levels cause the bzip2 algorithm to devote more resources to the compression effort, and may not result in a significant improvement over lower levels. If not specified, the default value is `1`.

Ice.ConsoleListener

Synopsis

`Ice.ConsoleListener=`*`num`* (.NET)

Description

If *`num`* is non-0, the Ice run time installs a `ConsoleTraceListener` that writes its messages to `stderr`. If *`num`* is 0, logging is disabled. Note that the setting of `#Ice.LogFile` overrides this property: if `Ice.LogFile` is set, messages are written to the log file regardless of the setting of `Ice.ConsoleListener`.

Ice.EventLog.Source

Synopsis

`Ice.EventLog.Source=`*`name`* (C++ & Windows only)

Description

Specifies the name of an event log source to be used by a Windows service that subclasses `Ice::Service`. The value of *`name`* represents a subkey of the `Eventlog` registry key. An application (or administrator) typically prepares the registry key when the service is installed. If no matching registry key is found, Windows logs events in the `Application` log. Any backslashes in *`name`* are silently converted to forward slashes. If not defined, `Ice::Service` uses the service name as specified by the `--service` option.

Ice.HTTPProxyHost

Synopsis

`Ice.HTTPProxyHost=`*`addr`*

Description

Specifies the host name or IP address of an HTTP proxy server. If *addr* is not empty, Ice uses the designated HTTP proxy server for all outgoing (client) connections.

Ice.HTTPProxyPort

Synopsis

`Ice.HTTPProxyPort=`*num*

Description

The port number of the HTTP proxy server. If not specified, the default value is `1080`.

Ice.ImplicitContext

Synopsis

`Ice.ImplicitContext=`*type*

Description

Specifies whether a communicator has an implicit request context and, if so, at what scope the context applies. Legal values for this property are `None` (equivalent to the empty string), `PerThread`, and `Shared`. If not specified, the default value is `None`.

> The `PerThread` type is currently not available for JavaScript or with C++ on Universal Windows (UWP).

Ice.LogFile

Synopsis

`Ice.LogFile=`*file*

Description

Replaces the communicator's default logger with a simple file-based logger implementation. This property does not affect the per-process logger. The logger creates the specified file if necessary, otherwise it appends to the file. If the logger is unable to open the file, the application receives an `InitializationException` during communicator initialization. If a logger object is supplied in the `InitializationData` argument during communicator initialization, it takes precedence over this property.

Ice.LogFile.SizeMax

Synopsis

`Ice.LogFile.SizeMax=`*num* `(C++)`

Description

num is a positive integer that represents the maximum size of log files configured through `Ice.LogFile`, in bytes. When a log file's size reaches *num*, the Ice file-based logger renames this log file to *baselogfilename-datetimestamp.ext* and creates a new log file. The default value for *num* is 0, which means that the log file's size is unlimited. In this case, the Ice file-based logger opens and writes to a single log file.

Ice.LogStdErr.Convert

Synopsis

`Ice.LogStdErr.Convert=`*`num`* (C++)

Description

If *`num`* is set to a value larger than 0, on Windows, the communicator's default logger converts log messages from the application's narrow string encoding (as defined by the installed narrow string converter, if any) to the Windows console's code page. The default value for this property is 1 when `Ice.StdErr` is not set, and 0 otherwise. This property is read by the first communicator created in a process; it is ignored by other communicators.

Ice.MessageSizeMax

Synopsis

`Ice.MessageSizeMax=`*`num`*

Description

This property controls the maximum size (in kilobytes) of an uncompressed protocol message that is accepted by the Ice run time. The size includes the size of the Ice protocol header. The default size is `1024` (`1` megabyte).

The only purpose of this property is to prevent a malicious or defective sender from triggering a large memory allocation in a receiver. If this is not a concern, you can set `Ice.MessageSizeMax` to 0; setting this property to 0 (or to a negative number) disables the message size limit altogether.

If the Ice run time in a receiver encounters an incoming message whose size exceeds the receiver's setting for `Ice.MessageSizeMax`, the run time raises a `MemoryLimitException` and closes the connection. For example, when a client receives an oversized reply message, the result of its invocation is a `MemoryLimitException`. When a server receives an oversized request message, the client receives a `ConnectionLostException` (because the server closed the connection) and the server logs a message if `Ice.Warn.Connections` is set.

Ice.Nohup

Synopsis

`Ice.Nohup=`*`num`* (C++, .NET)

Description

If *`num`* is set to a value larger than 0, the `Application` convenience class (as well as the `Ice::Service` class in C++) ignore `SIGHUP` on Unix and `CTRL_LOGOFF_EVENT` on Windows. As a result, an application that sets `Ice.Nohup` continues to run if the user that started the application logs off. The default value for `Application` is `0`, and the default value for `Ice::Service` is `1`.

Ice.NullHandleAbort

Synopsis

`Ice.NullHandleAbort=`*`num`* (C++, Objective-C, PHP, Python, Ruby)

Description

If *`num`* is set to a value larger than 0, invoking an operation using a null Ice C++98 smart pointer causes the program to abort immediately instead of raising `IceUtil::NullHandleException`.

> With the C++11 mapping, these smart pointers are used only in Ice-internal code. This property has no effect on `std::shared_ptr`.

Ice.Package.*module*

Synopsis

`Ice.Package.`*`module`*`=`*`package`* (Java, MATLAB)

Description

Ice for Java and Ice for MATLAB allow you to customize the packaging of generated code. If you use this feature, the Ice run time requires additional configuration in order to successfully unmarshal exceptions and concrete class types. This property associates a top-level Slice *`module`* with a Java or MATLAB *`package`*. If all top-level modules are generated into the same user-defined package, it is easier to use `Ice.Default.Package` instead.

Ice.PreloadAssemblies

Synopsis

`Ice.PrintProcessId=`*`num`* (.NET)

Description

If *`num`* is set to a value larger than 0, the Ice run-time will try to load all the assemblies referenced by the process during communicator initialization, otherwise the referenced assemblies will be initialized when the Ice run-time needs to lookup a C# class. The default value is 0.

Ice.PrintAdapterReady

Synopsis

`Ice.PrintAdapterReady=`*`num`*

Description

If *`num`* is set to a value larger than 0, an object adapter prints "*adapter_name* ready" on standard output after activation is complete. This is useful for scripts that need to wait until an object adapter is ready to be used.

Ice.PrintProcessId

Synopsis

`Ice.PrintProcessId=`*`num`*

Description

If *`num`* is set to a value larger than 0, the process ID is printed on standard output upon startup.

Ice.PrintStackTraces

Synopsis

`Ice.PrintStackTraces=`*`num`* (C++, JavaScript, Objective-C, PHP, Python, Ruby)

Description

(C++) If *`num`* is set to a value larger than 0, inserting an exception that derives from `Ice::Exception` into a logger helper class (such as `Ice::Warning`) also displays the exception's stack trace. Likewise, the `ice_stackTrace` function on the base exception class, `Ice::Exception`, will return the stack trace or an empty string depending on the value *num*. If not set, the default value depends on how the Ice run time is compiled: 0 for an optimized build and 1 for a debug build.

When `PrintStackTraces` is non-0, the constructor of `Ice::Exception` captures the current call stack, which adds a small overhead. This overhead is incurred even when the application never converts the captured stack into a stack trace.

On Windows, you need the Ice PDB files to obtain usable stack traces. If you build Ice from sources, the Ice build system always creates PDB files next to your DLLs and executables, and Windows will locate and use these PDB files. If you use Ice NuGet packages, follow the instructions in Using the Windows Binary Distributions.

This property also affects scripting languages that use the Ice for C++ run time (Objective-C, PHP, Python and Ruby). Enabling this property in these languages will only display C++ stack traces.

(JavaScript) If *num* is set to a value larger than 0, `Ice.Exception.toString` includes the exception's stack trace. The default value for *num* is 0.

Ice.ProgramName

Synopsis

`Ice.ProgramName=`*name*

Description

name is the program name, which is set automatically from `argv[0]` (C++) and from `AppDomain.CurrentDomain.FriendlyName` (.NET) during initialization. For Java, `Ice.ProgramName` is initialized to the empty string. The default name can be overridden by setting this property.

Ice.RetryIntervals

Synopsis

`Ice.RetryIntervals=`*num* `[`*num* `...]`

Description

This property defines the number of times an operation is automatically retried and the delay between each retry. For example, if the property is set to `0 100 500`, the operation is retried 3 times: immediately after the first failure, again after waiting 100ms after the second failure, and again after waiting 500ms after the third failure. The default value (`0`) means Ice retries once immediately. If set to `-1`, no retry occurs.

Ice.ServerIdleTime

Synopsis

`Ice.ServerIdleTime=`*num*

Description

If *num* is set to a value larger than 0, Ice automatically calls `Communicator::shutdown` once the communicator has been idle for *num* seconds. This shuts down the communicator's server side and causes all threads waiting in `Communicator::waitForShutdown` to return. After that, a server will typically do some clean-up work before exiting. The default value is 0, meaning that the server will not shut down automatically. This property is often used for servers that are automatically activated by IceGrid.

On Windows, the server idle time takes effect only once the server thread pool idle threads have been reaped (the thread idle time can be configured with the ThreadIdleTime thread pool property.

Ice.SOCKSProxyHost

Synopsis

`Ice.SOCKSProxyHost=`*`addr`*

Description

Specifies the host name or IP address of a SOCKS proxy server. If *`addr`* is not empty, Ice uses the designated SOCKS proxy server for all outgoing (client) connections.

> Ice currently only supports the SOCKS4 protocol, which means only IPv4 connections are allowed.

Ice.SOCKSProxyPort

Synopsis

`Ice.SOCKSProxyPort=`*`num`*

Description

The port number of the SOCKS proxy server. If not specified, the default value is `1080`.

Ice.StdErr

Synopsis

`Ice.StdErr=`*`filename`*

Description

If *`filename`* is not empty, the standard error stream of this process is redirected to this file, in append mode. This property is checked only for the first communicator that is created in a process.

Ice.StdOut

Synopsis

`Ice.StdOut=`*`filename`*

Description

If *`filename`* is not empty, the standard output stream of this process is redirected to this file, in append mode. This property is checked only for the first communicator created in a process.

Ice.SyslogFacility

Synopsis

`Ice.SyslogFacility=`*`string`* (Unix only)

Description

This property sets the syslog facility to *`string`*. This property has no effect if `Ice.UseSyslog` is not set.

`string` can be any of syslog facilities: `LOG_AUTH, LOG_AUTHPRIV, LOG_CRON, LOG_DAEMON, LOG_FTP, LOG_KERN, LOG_LOCAL0, LOG_LOCAL1, LOG_LOCAL2, LOG_LOCAL3, LOG_LOCAL4, LOG_LOCAL5, LOG_LOCAL6, LOG_LOCAL7,`

`LOG_LPR, LOG_MAIL, LOG_NEWS, LOG_SYSLOG, LOG_USER, LOG_UUCP`.

The default value is `LOG_USER`.

Ice.ThreadInterruptSafe

Synopsis

`Ice.ThreadInterruptSafe=`*`num`* (Java)

Description

If *`num`* is set to a value larger than 0, Ice for Java disables message caching by setting `Ice.CacheMessageBuffers` to 0 and takes other steps necessary to ensure that Java interrupts work correctly. If not defined, the default value is 0.

Ice.ToStringMode

Synopsis

`Ice.ToStringMode=`*`string`*

Description

`string` must be one of the following: `Unicode, ASCII, Compat.`

This property maps to an enumerator of `Ice::ToStringMode` and controls how `identityToString` and `proxyToString` on the `Communicator` local interface escape non-printable ASCII characters and non-ASCII characters.

The default value is `Unicode`.

Ice.UseApplicationClassLoader

Synopsis

`Ice.UseApplicationClassLoader=`*`num`* (Java)

Description

If *`num`* is set to a value larger than 0, when dispatching an operation to a servant, Ice sets the context class loader to the class loader of the servant class. With the Java-Compat mapping only, it also sets the context class loader when calling AMI callbacks (with the class loader of the AMI callback class).

Ice.UseSyslog

Synopsis

`Ice.UseSyslog=`*`num`* (Unix only)

Description

If *`num`* is set to a value larger than 0, a special logger is installed that logs to the `syslog` service instead of standard error. The identifier for `syslog` is the value of `Ice.ProgramName`. Use `Ice.SyslogFacility` to select a `syslog` facility.

Glacier2.*

On this page:

Glacier2.AddConnectionContext

Synopsis

`Glacier2.AddConnectionContext=`*`num`*

Description

If *`num`* is set to 1 or 2, Glacier2 adds a number of key-value pairs to the request context that it sends with each request. If *`num`* is set to 1, these entries are added to the context for all forwarded requests. If *`num`* is set to 2, the contexts are added only to calls to `checkPermissions` and `authorize` on permission verifiers, and to calls to `create` on session managers.

If num is non-0, Glacier2 adds the following context entries:

`_con.type`	The type of the connection as returned by `Connection::type`.
`_con.localAddress`	The local address (TCP and SSL only).
`_con.localPort`	The local port (TCP and SSL only).
`_con.remoteAddress`	The remote address (TCP and SSL only).
`_con.remotePort`	The remote port (TCP and SSL only).

`_con.cipher`	The cipher (SSL only).
`_con.peerCert`	The first certificate of the client certificate chain (SSL only).

The default value is 0, meaning that no contexts are added.

Glacier2.AddUserToAllowCategories

Synopsis

`Glacier2.AddUserToAllowCategories=`*`num`*

Description

Specifies whether to add an authenticated user ID to the `Glacier2.AllowCategories` property when creating a new session. The legal values are shown below:

0	Do not add the user ID (default).
1	Add the user ID.
2	Add the user ID with a leading underscore.

This property is deprecated and supported only for backward-compatibility. New applications should use `Glacier2.Filter.Category.AcceptUser`.

Glacier2.Admin.*AdapterProperty*

Synopsis

`Glacier2.Admin.`*`AdapterProperty`*`=`*`value`*

Description

Glacier2 uses the adapter name `Glacier2.Admin` for its administrative object adapter. Therefore, adapter properties can be used to configure this adapter.

The `Glacier2.Admin.Endpoints` property must be defined to enable the administrative object adapter.

Glacier2's administrative interface allows a remote client to shut down the router; we generally recommend the use of endpoints that are accessible only from behind a firewall.

Glacier2.AllowCategories

Synopsis

`Glacier2.AllowCategories=`*`list`*

Description

Specifies a white space-separated list of identity categories. If this property is defined, then the Glacier2 router only allows requests to Ice objects with an identity that matches one of the categories from this list. If `Glacier2.AddUserToAllowCategories` is defined with a non-0 value, the router automatically adds the user ID of each session to this list.

This property is deprecated and supported only for backward-compatibility. New applications should use `Glacier2.Filter.Category.Accept`.

Glacier2.Client.*AdapterProperty*

Synopsis

`Glacier2.Client.`*`AdapterProperty`*`=`*`value`*

Description

Glacier2 uses the adapter name `Glacier2.Client` for the object adapter that it provides to clients. Therefore, adapter properties can be used to configure this adapter.

This adapter must be accessible to clients of Glacier2. Use of a secure transport for this adapter is highly recommended.

Note that `Glacier2.Registry.Client.Endpoints` controls the client endpoint for Glacier2. The port numbers 4063 (for TCP) and 4064 (for SSL) are reserved for Glacier2 by the Internet Assigned Numbers Authority (IANA).

Glacier2.Client.AlwaysBatch

Synopsis

`Glacier2.Client.AlwaysBatch=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router always batches queued oneway requests from clients to servers regardless of the value of their `_fwd` contexts. This property is only relevant when `Glacier2.Client.Buffered` is enabled. The default value is `0`.

Glacier2.Client.Buffered

Synopsis

`Glacier2.Client.Buffered=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router operates in buffered mode, in which incoming requests from clients are queued and processed in a separate thread. If *`num`* is set to 0, the router operates in unbuffered mode in which a request is forwarded in the same thread that received it. The default value is `1`.

Glacier2.Client.ForwardContext

Synopsis

`Glacier2.Client.ForwardContext=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router includes the request context when forwarding requests from clients to servers. The default value is `0`.

Glacier2.Client.SleepTime

Synopsis

`Glacier2.Client.SleepTime=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router sleeps for the specified number of milliseconds after forwarding all queued requests

from a client. This delay is useful for batched delivery because it makes it more likely for events to accumulate in a single batch. Similarly, if overrides are used, the delay makes it more likely for overrides to actually take effect. This property is only relevant when `Glacier2.Client.Buffered` is enabled. The default value is `0`.

Glacier2.Client.Trace.Override

Synopsis

`Glacier2.Client.Trace.Override=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router logs a trace message whenever a request was overridden. The default value is `0`.

Glacier2.Client.Trace.Reject

Synopsis

`Glacier2.Client.Trace.Reject=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router logs a trace message whenever the router's configured filters reject a client's request. The default value is `0`.

Glacier2.Client.Trace.Request

Synopsis

`Glacier2.Client.Trace.Request=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router logs a trace message for each request that is forwarded from a client. The default value is `0`.

Glacier2.CryptPasswords

Synopsis

`Glacier2.CryptPasswords=`*`file`*

Description

Specifies the file name of a Glacier2 access control list. Each line of the file must contain a user name and an encrypted password, separated by white space, as described in Writing a Password File. This property is ignored if `Glacier2.PermissionsVerifier` is defined.

Glacier2.Filter.AdapterId.Accept

Synopsis

`Glacier2.Filter.AdapterId.Accept=`*`list`*

Description

Specifies a space-separated list of adapter identifiers. If defined, the Glacier2 router filters requests so that it only allows requests to Ice

objects with an adapter identifier that matches one of the entries in this list.

Identifiers that contain spaces must be enclosed in single or double quotes. Single or double quotes that appear within an identifier must be escaped with a leading backslash.

Glacier2.Filter.Address.Accept

Synopsis

`Glacier2.Filter.Address.Accept=`*`list`*

Description

Specifies a space-separated list of address-port pairs. When defined, the Glacier2 router filters requests so that it only allows requests to Ice objects through proxies that contain network endpoint information that matches an address-port pair listed in this property. If not defined, the default value is *, which indicates that any network address is permitted. Requests accepted by this property may be rejected by the `Glacier2.Filter.Address.Reject` property.

Each pair is of the form *`address`*`:`*`port`*. The *`address`* or *`port`* number portion can include wildcards ('*') or value ranges or groups. Ranges and groups have the form `[`*`value1, value2, value3, ...`*`]` and/or `[`*`value1-value2`*`]`. Wildcards, ranges, and groups may appear anywhere in the address-port pair string.

Glacier2.Filter.Address.Reject

Synopsis

`Glacier2.Filter.Address.Reject=`*`list`*

Description

Specifies a space-separated list of address-port pairs. When defined, the Glacier2 router rejects requests to Ice objects through proxies that contain network endpoint information that matches an address-port pair listed in this property. If not set, the Glacier2 router allows requests to any network address unless the `Glacier2.Filter.Address.Accept` property is set, in which case requests will be accepted or rejected based on the `Glacier2.Filter.Address.Accept` property. If both the `Glacier2.Filter.Address.Accept` and `Glacier2.Filter.Address.Reject` properties are defined, the `Glacier2.Filter.Address.Reject` property takes precedence.

Each pair is of the form *`address`*`:`*`port`*. The *`address`* or *`port`* number portion can include wildcards ('*') or value ranges or groups. Ranges and groups have the form `[`*`value1, value2, value3, ...`*`]` and/or `[`*`value1-value2`*`]`. Wildcards, ranges, and groups may appear anywhere in the address-port pair string.

Glacier2.Filter.Category.Accept

Synopsis

`Glacier2.Filter.Category.Accept=`*`list`*

Description

Specifies a space-separated list of identity categories. If defined, the Glacier2 router filters requests so that it only allows requests to Ice objects with an identity that matches one of the categories in this list. If `Glacier2.Filter.Category.AcceptUser` is defined with a non-0 value, the router automatically adds the user name of each session to this list.

Categories that contain spaces must be enclosed in single or double quotes. Single or double quotes that appear within a category must be escaped with a leading backslash.

Glacier2.Filter.Category.AcceptUser

Synopsis

`Glacier2.Filter.Category.AcceptUser=`*`num`*

Description

Specifies whether to add an authenticated user ID to the `Glacier2.Filter.Category.Accept` property when creating a new session. The legal values are shown below:

0	Do not add the user ID (default).
1	Add the user ID.
2	Add the user ID with a leading underscore.

Glacier2.Filter.Identity.Accept

Synopsis

`Glacier2.Filter.Identity.Accept=`*`list`*

Description

Specifies a space-separated list of identities. If defined, the Glacier2 router filters requests so that it only allows requests to Ice objects with an identity that matches one of the entries in this list.

Identities that contain spaces must be enclosed in single or double quotes. Single or double quotes that appear within an identity must be escaped with a leading backslash.

Glacier2.Filter.ProxySizeMax

Synopsis

`Glacier2.Filter.ProxySizeMax=`*`num`*

Description

If set, the Glacier2 router rejects requests whose stringified proxies are longer than *`num`*. This helps secure the system against attack. If not set, Glacier2 will accept requests using proxies of any length.

Glacier2.InstanceName

Synopsis

`Glacier2.InstanceName=`*`name`*

Description

Specifies a default identity category for the Glacier2 objects. If defined, the identity of the Glacier2 administrative interface becomes *`name`*`/admin` and the identity of the Glacier2 router interface becomes *`name`*`/router`.

If not defined, the default value is `Glacier2`.

Glacier2.PermissionsVerifier

Synopsis

`Glacier2.PermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::PermissionsVerifier` interface for controlling access to Glacier2 sessions. The router invokes this proxy to validate the user name and password of each new session. Sessions created from a secure

connection are verified by the object specified in `Glacier2.SSLPermissionsVerifier`. For simple configurations, you can specify the name of a password file using `Glacier2.CryptPasswords`.

Glacier2 supplies a "null" permissions verifier object that accepts any username and password combination for situations in which no authentication is necessary. To enable this verifier, set the property value to *`instance`*`/NullPermissionsVerifier`, where *`instance`* is the value of `Glacier2.InstanceName`.

As a proxy property, you can configure additional aspects of the proxy using properties.

Glacier2.ReturnClientProxy

Synopsis

`Glacier2.ReturnClientProxy=`*`num`*

Description

If *`num`* is a value greater than 0, Glacier2 maintains backward compatibility with clients using Ice versions prior to 3.2.0. In this case you should also define `Glacier2.Client.PublishedEndpoints` to specify the endpoints that clients should use to contact the router. For example, if the Glacier2 router resides behind a network firewall, the `Glacier2.Client.PublishedEndpoints` property should specify the firewall's external endpoints.

If not defined, the default value is 0.

Glacier2.RoutingTable.MaxSize

Synopsis

`Glacier2.RoutingTable.MaxSize=`*`num`*

Description

This property sets the size of the router's routing table to *`num`* entries. If more proxies are added to the table than this value, proxies are evicted from the table on a least-recently used basis.

Clients based on Ice version 3.1 and later automatically retry operation calls on evicted proxies and transparently re-add such proxies to the table. Clients based on Ice versions earlier than 3.1 receive an `ObjectNotExistException` for invocations on evicted proxies. For such older clients, *`num`* must be set to a sufficiently large value to prevent these clients from failing.

The default size of the routing table is 1000.

Glacier2.Server.*AdapterProperty*

Synopsis

`Glacier2.Server.`*`AdapterProperty=value`*

Description

Glacier2 uses the adapter name `Glacier2.Server` for the object adapter that it provides to servers. Therefore, adapter properties can be used to configure this adapter.

This adapter provides access to the `SessionControl` interface and must be accessible to servers that call back to router clients.

Glacier2.Server.AlwaysBatch

Synopsis

`Glacier2.Server.AlwaysBatch=`*`num`*

Description

If *num* is set to a value larger than 0, the Glacier2 router always batches queued oneway requests from servers to clients regardless of the value of their `_fwd` contexts. This property is only relevant when `Glacier2.Server.Buffered` is enabled. The default value is `0`.

Glacier2.Server.Buffered

Synopsis

`Glacier2.Server.Buffered=`*num*

Description

If *num* is set to a value larger than 0, the Glacier2 router operates in buffered mode, in which incoming requests from servers are queued and processed in a separate thread. If *num* is set to 0, the router operates in unbuffered mode in which a request is forwarded in the same thread that received it. The default value is `1`.

Glacier2.Server.ForwardContext

Synopsis

`Glacier2.Server.ForwardContext=`*num*

Description

If *num* is set to a value larger than 0, the Glacier2 router includes the request context when forwarding requests from servers to clients. The default value is `0`.

Glacier2.Server.SleepTime

Synopsis

`Glacier2.Server.SleepTime=`*num*

Description

If *num* is set to a value larger than 0, the Glacier2 router sleeps for the specified number of milliseconds after forwarding all queued requests from a server. This delay is useful for batched delivery because it makes it more likely for events to accumulate in a single batch. Similarly, if overrides are used, the delay makes it more likely for overrides to actually take effect. This property is only relevant when `Glacier2.Server.Buffered` is enabled. The default value is `0`.

Glacier2.Server.Trace.Override

Synopsis

`Glacier2.Server.Trace.Override=`*num*

Description

If *num* is set to a value larger than 0, the Glacier2 router logs a trace message whenever a request is overridden. The default value is `0`.

Glacier2.Server.Trace.Request

Synopsis

`Glacier2.Server.Trace.Request=`*num*

Description

If *num* is set to a value larger than 0, the Glacier2 router logs a trace message for each request that is forwarded from a server. The default value is `0`.

Glacier2.SessionManager

Synopsis

`Glacier2.SessionManager=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SessionManager` interface. The router invokes this proxy to create a new session for a client, but only after the router validates the client's user name and password.

As a proxy property, you can configure additional aspects of the proxy using properties.

Glacier2.SessionTimeout

Synopsis

`Glacier2.SessionTimeout=`*`num`*

Description

If *num* is set to a value larger than 0, a client's session with the Glacier2 router expires after the specified *num* seconds of inactivity. The default value is `0`, meaning sessions do not expire due to inactivity.

It is important to choose *num* such that client sessions do not expire prematurely.

The session timeout may also affect Active Connection Management (ACM) for client connections, which are connections to the router's object adapter named `Glacier2.Client`. If you have not explicitly configured the router with a value for `Glacier2.Client.ACM.Timeout`, the router uses the session timeout as the value for this property. If no session timeout is defined, the router's incoming client connections use its default ACM behavior.

Glacier2.SSLPermissionsVerifier

Synopsis

`Glacier2.SSLPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLPermissionsVerifier` interface for controlling access to Glacier2 sessions. The router invokes this proxy to verify the credentials of clients that attempt to create a session from a secure connection. Sessions created with a user name and password are verified by the object specified in `Glacier2.PermissionsVerifier`.

Glacier2 supplies a "null" permissions verifier object that accepts the credentials of any client for situations in which no authentication is necessary. To enable this verifier, set the property value to *`instance`*`/NullSSLPermissionsVerifier`, where *`instance`* is the value of `Glacier2.InstanceName`.

As a proxy property, you can configure additional aspects of the proxy using properties.

Glacier2.SSLSessionManager

Synopsis

`Glacier2.SSLSessionManager=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLSessionManager` interface for managing sessions. The router invokes this proxy to create a new session for a client that has called `createSessionFromSecureConnection`.

As a proxy property, you can configure additional aspects of the proxy using properties.

Glacier2.Trace.RoutingTable

Synopsis

`Glacier2.Trace.RoutingTable=`*`num`*

Description

The routing table trace level:

0	No routing table trace (default).
1	Logs a message for each proxy that is added to the routing table.
2	Logs a message for each proxy that is evicted from the routing table (see `#Glacier2.RoutingTable.MaxSize`).
3	Combines the output for trace levels 1 and 2.

Glacier2.Trace.Session

Synopsis

`Glacier2.Trace.Session=`*`num`*

Description

If *`num`* is set to a value larger than 0, the Glacier2 router logs trace messages about session-related activities. The default value is `0`.

Ice.ACM.*

On this page:

Ice.ACM.Close

Synopsis

`Ice.ACM.Close=`*`num`*

Description

This property determines when Active Connection Management (ACM) closes a connection. Legal values for *num* correspond directly to the `ACMClose` enumerators found in `Connection.ice`:

Slice

```
local enum ACMClose
{
    CloseOff,
    CloseOnIdle,
    CloseOnInvocation,
    CloseOnInvocationAndIdle,
    CloseOnIdleForceful
}
```

The table below describes the semantics of each value:

Enumerator	Property Value	Description
`CloseOff`	0	Disables automatic connection closure. A connection will be closed when the communicator is destroyed, when there's a network failure or when the peer closes it.
`CloseOnIdle`	1	Gracefully closes a connection that has been idle for the configured timeout period. A connection is considered to be idle if it has not sent or received any messages, and is not waiting for the completion of any outgoing invocations or incoming requests.
`CloseOnInvocation`	2	Forcefully closes a connection with pending outgoing invocations that has otherwise been idle for the configured timeout period, meaning it has not sent or received any messages, and is not waiting for the completion of any incoming requests. This is useful when you don't want an idle connection to be closed but still want the connection to be forcefully closed when outgoing invocations are in progress and no messages are received from the server, potentially indicating that the server is dead. This setting should be used with a heartbeat configuration where heartbeats are sent at regular intervals by a server while a request is being dispatched (such as `HeartbeatOnDispatch`, `HeartbeatOnIdle` or `HeartbeatAlways`). Hearbeat messages prevent a connection from becoming idle, and the sudden lack of heartbeat messages from a server typically indicate a problem in the server.

`CloseOnInvocationAndIdle`	3	Combines the behaviors of `CloseOnIdle` and `CloseOnInvocation`, meaning an idle connection is closed gracefully, while an otherwise idle connection with pending outgoing invocations is closed forcefully.
`CloseOnIdleForceful`	4	Forcefully closes a connection that has been idle for the configured timeout period, regardless of whether the connection has pending outgoing invocations or incoming requests. This is typically used together with a heartbeat configuration that keeps idle connections alive.

You must set a non-zero value for `Ice.ACM.Timeout` for this property to have any effect. Use `Ice.ACM.Close` in conjunction with `Ice.ACM.Heartbeat` to tailor a connection management policy for your application. Use the corresponding `Ice.ACM.Client.Close` or `Ice.ACM.Server.Close` properties to override this setting in a client or server context, respectively. See Connection Closure for more information on closing connections.

If not defined, the default value is 3 in a client context and 2 in a server context.

Ice.ACM.Heartbeat

Synopsis

`Ice.ACM.Heartbeat=`*`num`*

Description

This property determines whether heartbeats are enabled. You can use heartbeats in conjunction with the `Ice.ACM.Close` and `Ice.ACM.Timeout` properties to create a connection management policy as part of Active Connection Management (ACM). Legal values for *`num`* correspond directly to the `ACMHeartbeat` enumerators found in `Connection.ice`:

Slice

```
local enum ACMHeartbeat
{
    HeartbeatOff,
    HeartbeatOnDispatch,
    HeartbeatOnIdle,
    HeartbeatAlways
}
```

The table below describes the semantics of each value:

Enumerator	Property Value	Description
`HeartbeatOff`	0	Disables heartbeats.
`HeartbeatOnDispatch`	1	Send a heartbeat at regular intervals if the connection is idle and only if there are pending dispatch. This is useful if you don't want to send heartbeats when the connection is idle but still want the connection to notify the client that it's alive while the dispatch are in progress.
`HeartbeatOnIdle`	2	Send a heartbeat at regular intervals when the connection is idle.
`HeartbeatAlways`	3	Send a heartbeat at regular intervals until the connection is closed.

You must set a non-zero value for `Ice.ACM.Timeout` for this property to have any effect. Use the corresponding `Ice.ACM.Client.Heartbeat` or `Ice.ACM.Server.Heartbeat` properties to override this setting in a client or server context, respectively.

To send a heartbeat, the Ice run time sends a validate connection message to the remote end. Heartbeats are not supported on datagram connections.

If not defined, the default value is `1`.

Ice.ACM.Timeout

Synopsis

`Ice.ACM.Timeout=`*`num`*

Description

The value for *`num`* represents a timeout in seconds for Active Connection Management (ACM). The setting for `Ice.ACM.Close` determines when connections are automatically closed, and the setting for `Ice.ACM.Heartbeat` can be used to prevent connections from being closed prematurely. Use the corresponding `Ice.ACM.Client.Timeout` or `Ice.ACM.Server.Timeout` properties to override this setting in a client or server context, respectively. The value must be positive or 0. Setting the timeout to 0 disables ACM which is equivalent to setting `CloseOff` and `HeartbeatOff` for the close and heartbeat ACM settings respectively.

If not defined, the default value is `60`.

Ice.ACM.Client.Close

Synopsis

`Ice.ACM.Client.Close=`*`num`*

Description

Overrides the value of `Ice.ACM.Close` in a client context, meaning for outgoing connections.

Ice.ACM.Client.Heartbeat

Synopsis

`Ice.ACM.Client.Heartbeat=`*`num`*

Description

Overrides the value of `Ice.ACM.Heartbeat` in a client context, meaning for outgoing connections.

Ice.ACM.Client.Timeout

Synopsis

`Ice.ACM.Client.Timeout=`*`num`*

Description

Overrides the value of `Ice.ACM.Timeout` in a client context, meaning for outgoing connections.

Ice.ACM.Server.Close

Synopsis

`Ice.ACM.Server.Close=`*`num`*

Description

Overrides the value of `Ice.ACM.Close` in a server context, meaning for incoming connections.

Ice.ACM.Server.Heartbeat

Synopsis

`Ice.ACM.Server.Heartbeat=`*`num`*

Description

Overrides the value of `Ice.ACM.Heartbeat` in a server context, meaning for incoming connections.

Ice.ACM.Server.Timeout

Synopsis

`Ice.ACM.Server.Timeout=`*`num`*

Description

Overrides the value of `Ice.ACM.Timeout` in a server context, meaning for incoming connections.

Ice.Admin.*

On this page:

Ice.Admin.*AdapterProperty*

Synopsis

`Ice.Admin.`*`AdapterProperty`*`=`*`value`*

Description

The Ice run time creates and activates an administrative object adapter named `Ice.Admin` if the Administrative Facility is enabled, `Ice.Admin.Endpoints` is defined and one of the following are true:

- `Ice.Admin.DelayCreation` is not enabled
- `Ice.Admin.DelayCreation` is enabled and the application calls `getAdmin` on the communicator after communicator initialization
- the application calls `createAdmin` with a null `adminAdapter` parameter

This object adapter is created to host the admin object. Adapter properties can be used to configure the `Ice.Admin` object adapter.

Note that enabling the `Ice.Admin` object adapter is a security risk because a hostile client could use the administrative object to shut down the process. As a result, the endpoints for this object adapter should be carefully defined so that only trusted clients are allowed to use it.

Ice.Admin.DelayCreation

Synopsis

`Ice.Admin.DelayCreation=`*`num`*

Description

If *`num`* is a value greater than zero, the Ice run time delays the creation of the `Ice.Admin` administrative object adapter until `getAdmin` is invoked on the communicator. If not specified, the default value is zero, meaning the `Ice.Admin` object adapter is created immediately after all plug-ins are initialized, provided `Ice.Admin.Endpoints` is defined.

Ice.Admin.Enabled

Synopsis

`Ice.Admin.Enabled=`*`num`*

Description

If *`num`* is a value greater than zero, the Administrative Facility is enabled. If *`num`* is a zero or a negative value, the Administrative Facility is disabled. If this property is not set at all, the Administrative Facility is enabled when Ice.Admin.Endpoints is defined and not empty, and is disabled otherwise.

Ice.Admin.Facets

Synopsis

`Ice.Admin.Facets=`*`name`* `[`*`name`* `...]`

Description

Specifies the facets enabled by the administrative object, allowing you to filter the facets that the administrative object enables by default. Facet names are delimited by commas or white space. A facet name that contains white space must be enclosed in single or double quotes. If not specified, all facets are enabled. While the Ice run time creates only the built-in facets (such as Process and Properties) that are enabled, you can create administrative facets without checking the value of this property. Ice ensures that only enabled administrative facets are available to clients.

Ice.Admin.InstanceName

Synopsis

`Ice.Admin.InstanceName=`*`name`*

Description

Specifies an identity category for the administrative object, when this object is created during communicator initialization or by a call to `getAdmin` on the communicator. If defined, the identity of the object becomes *`name`*`/admin`. If not specified, the default identity category is a UUID.

Ice.Admin.Logger.KeepLogs

Synopsis

`Ice.Admin.Logger.KeepLogs=`*`num`*

Description

The `Logger` admin facet, when enabled, caches up the *num* most recent log messages with a type other than `Ice::TraceMessage`. When *num* is 0 or less than 0, the Logger facet does not cache any of these log messages. The default value for *num* is 100.

Ice.Admin.Logger.KeepTraces

Synopsis

`Ice.Admin.Logger.KeepTraces=`*`num`*

Description

The `Logger` admin facet, when enabled, caches up the *num* most recent log messages with type `Ice::TraceMessage`. When *num* is 0 or less than 0, the Logger facet does not cache any of these trace messages. The default value for *num* is 100.

Ice.Admin.Logger.Properties

Synopsis

`Ice.Admin.Logger.Properties=`*`propertyList`*

Description

The `Logger` admin facet, when enabled, creates its own communicator to send log messages to attached remote loggers. Without this sub-communicator, sending log messages to remote loggers could trigger more local logging, which in turn would generate more logs sent to

remote loggers: a single genuine log could trigger an infinite number of log messages.

The properties of this sub-communicator are a few properties of the application's communicator (`Ice.Default.Locator`, `Ice.Plugin.IceSSL` and all `IceSSL` properties), plus the properties (if any) specified by *propertyList*. *propertyList* is a sequence of strings, that Ice reads using `getPropertyAsList`. Each of these strings uses the syntax *PropertyName=PropertyValue* to set a property. For example, you could turn on protocol tracing on the `Logger` facet's sub-communicator with:

```
Ice.Admin.Logger.Properties=Ice.Trace.Protocol=1
```

Ice.Admin.ServerId

Synopsis

`Ice.Admin.ServerId=`*`id`*

Description

Specifies an identifier that uniquely identifies the process when the Ice run time registers the Process facet of its admin object with the locator registry.

Ice.Config

Ice.Config

Synopsis

```
--Ice.Config
--Ice.Config=1
--Ice.Config=config_file[,config_file,...]
```

Description

This property must be set from the command line with one of the options `--Ice.Config`, `--Ice.Config=1`, or `--Ice.Config=`*`config_file`*.

If the `Ice.Config` property is empty or set to 1, or not set at all, the Ice run time examines the contents of the `ICE_CONFIG` environment variable to retrieve the path names of one or more configuration files. Otherwise, `Ice.Config` must be set to the path names of one or more configuration files, separated by commas (path names can be relative or absolute). Property values are read from each of the configuration files listed.

In Java, Ice first attempts to open a configuration file as a class loader resource. If that attempt fails, Ice opens the configuration file in the local file system.

Configuration files use a simple syntax consisting of *name=value* pairs with support for comments and escaping.

See Also

- Using Configuration Files
- Configuration File Syntax
- Alternate Property Stores

Ice.Default.*

On this page:

Ice.Default.CollocationOptimized

Synopsis

`Ice.Default.CollocationOptimized=`*`num`*

Description

Specifies whether proxy invocations use collocation optimization by default. When enabled, proxy invocations on a collocated servant (i.e., a servant whose object adapter was created by the same communicator as the proxy) are made more efficiently by avoiding the network stack.

If not specified, the default value is 1. Set the property to 0 to disable collocation optimization by default.

Ice.Default.EncodingVersion

Synopsis

`Ice.Default.EncodingVersion=`*`ver`*

Description

If this property is not defined, Ice 3.5 uses encoding version 1.1 by default. To force the Ice run time to use encoding version 1.0 as the default instead, set this property to `1.0`:

```
Ice.Default.EncodingVersion=1.0
```

Ice.Default.EndpointSelection

Synopsis

`Ice.Default.EndpointSelection=`*`policy`*

Description

This property controls the default endpoint selection policy for proxies with multiple endpoints. Permissible values are `Ordered` and `Random`. The default value of this property is `Random`.

Ice.Default.Host

Synopsis

`Ice.Default.Host=`*`host`*

Description

If an endpoint does not specify a host name (i.e., omits the `-h` *`host`* option in IP-based endpoints or the `-a` *`address`* option in a Bluetooth endpoint), the *`host`* value from this property is used instead. The property has no default value.

Ice.Default.InvocationTimeout

Synopsis

`Ice.Default.InvocationTimeout=`*`num`*

Description

Specifies the default invocation timeout in milliseconds to use for all proxies. If not defined, the default timeout is `-1`, which means an invocation never times out.

If set to `-2`, invocation timeouts are disabled and the Ice run time behaves like Ice versions < 3.6: it uses connection timeouts (defined on the endpoints) to wait for the response of the invocation. This setting is provided only for backward compatibility and might be deprecated in a future Ice major release.

Ice.Default.Locator

Synopsis

`Ice.Default.Locator=`*`locator`*

Description

Specifies a default locator for all proxies and object adapters. The value is a stringified proxy for the IceGrid locator object. The default locator can be overridden on a proxy using the `ice_locator` proxy method. The default value is no locator.

The default identity of the IceGrid locator object is `IceGrid/Locator`, but this identity is influenced by the `IceGrid.InstanceName` property. The locator object is available on the IceGrid client endpoints. For example, suppose `IceGrid.Registry.Client.Endpoints` is set as follows:

```
IceGrid.Registry.Client.Endpoints=tcp -p 12000 -h localhost
```

In this case, the stringified proxy for the IceGrid locator is:

```
Ice.Default.Locator=IceGrid/Locator:tcp -p 12000 -h localhost
```

As a proxy property, you can configure additional aspects of the proxy using properties.

Ice.Default.LocatorCacheTimeout

Synopsis

`Ice.Default.LocatorCacheTimeout=`*`num`*

Description

Specifies the default locator cache timeout for indirect proxies. If *`num`* is set to a value larger than 0, locator cache entries older than *`num`* seconds are ignored. If set to 0, the locator cache is not used. If set to `-1`, locator cache entries do not expire.

Once a cache entry has expired, the Ice run time performs a new locate request to refresh the cache before sending the next invocation; therefore, the invocation is delayed until the run time has refreshed the entry. If you set `Ice.BackgroundLocatorCacheUpdates` to a non-0value, the lookup to refresh the cache is still performed but happens in the background; this avoids the delay for the first invocation that follows expiry of a cache entry.

Ice.Default.Package

Synopsis

`Ice.Default.Package=`*`package`*

Description

Ice for Java and Ice for MATLAB allow you to customize the packaging of generated code. If you use this feature, the Ice run time requires additional configuration in order to successfully unmarshal exceptions and concrete class types. This property specifies a default package to use if other attempts by the Ice run time to dynamically load a generated class have failed. Also see `Ice.Package.`*`module`*.

Ice.Default.PreferSecure

Synopsis

`Ice.Default.PreferSecure=`*`num`*

Description

Specifies whether secure endpoints are given precedence in proxies by default. The default value of *`num`* is 0, meaning that insecure endpoints are given preference.

Setting this property to a non-0 value is the equivalent of invoking the proxy method `ice_preferSecure(true)` on proxies created by the Ice run time, such as those returned by `stringToProxy` or received as the result of an invocation. Proxies created by methods such as `ice_oneway` inherit the setting of the original proxy. If you want to force all proxies to use only secure endpoints, use `Ice.Override.Secure` instead.

See Configuring Secure Proxies for a discussion of secure proxies.

Ice.Default.Protocol

Synopsis

`Ice.Default.Protocol=`*`protocol`*

Description

Sets the protocol that is being used if an endpoint uses default as the protocol specification. The default value is `tcp`.

Ice.Default.Router

Synopsis

`Ice.Default.Router=`*`router`*

Description

Specifies the default router for all proxies. The value is a stringified proxy for the Glacier2 router control interface. The default router can be overridden on a proxy using the `ice_router` proxy method. The default value is no router. This property is only for proxies: it does not add a router to object adapters.

As a proxy property, you can configure additional aspects of the proxy using properties.

Ice.Default.SlicedFormat

Synopsis

`Ice.Default.SlicedFormat=`*`num`*

Description

Specifies the encoding format of Slice classes and exceptions. The default value of *`num`* is 0, meaning that the encoding uses the compact format. Set this property to a non-0 value to use the sliced format by default. This setting is only relevant when using version 1.1 of the Ice encoding.

Note that you can also specify whether certain operations use the sliced format by annotating their definitions with metadata.

Ice.Default.SourceAddress

Synopsis

`Ice.Default.SourceAddress=`*`addr`*

Description

If specified, outgoing socket connections will be bound using the given address *`addr`*. This allows to set a specific IP address as the source address of IP packets but it doesn't necessarily imply that the operating system will use the network interface matching this IP address to send out the IP packet. It must be set to a numeric IP address. Proxy endpoints can override this setting with the `--sourceAddress` option. If no source address is configured, the Ice run time uses the operating system's default behavior for binding an outgoing socket connection.

> This feature is not supported on WinRT.

Ice.Default.Timeout

Synopsis

`Ice.Default.Timeout=`*`num`*

Description

Specifies the default timeout in milliseconds to use for an endpoint if no timeout is configured using the `-t` option. Use the value `-1` to configure an infinite default timeout. If not defined, the default timeout is `60000`.

Ice.InitPlugins

Ice.InitPlugins

Synopsis

`Ice.InitPlugins=`*`num`*

Description

If *`num`* is a value greater than zero, the Ice run time automatically initializes the plug-ins it has loaded. The order in which plug-ins are loaded and initialized is determined by `Ice.PluginLoadOrder`. An application may need to set this property to zero in order to interact directly with a plug-in after it has been loaded but before it is initialized. In this case, the application must invoke `initializePlugins` on the plug-in manager to complete the initialization process. If not defined, the default value is 1.

> When the admin Object is created, its `Process` facet is registered with the configured Locator registry if `Ice.Admin.ServerId` is set. This remote registration call may depend on plug-ins, such as the IceSSL plug-in. As a result, if you delay the initialization of the plug-ins by setting `Ice.InitPlugins` to 0, you may also want to delay the creation of the admin Object by setting `Ice.Admin.DelayCreation` to 1.

Ice.IPv4

Ice.IPv4

Synopsis

`Ice.IPv4=`*`num`*

Description

Specifies whether Ice uses IPv4. If *`num`* is a value greater than zero, IPv4 is enabled. If not specified, the default value is 1.

Ice.IPv6

Ice.IPv6

Synopsis

`Ice.IPv6=`*`num`*

Description

Specifies whether Ice uses IPv6. If *`num`* is a value greater than zero, IPv6 is enabled. If not specified, the default value is 1 if the system supports the creation of IPv6 sockets, and 0 otherwise.

Platform Notes

Java

Java's default network stack always accepts both IPv4 and IPv6 connections regardless of the settings of `Ice.IPv6`. You can configure the Java run time to use only IPv4 by starting your application with the following JVM option:

Java

```
java -Djava.net.preferIPv4Stack=true ...
```

Ice.Override.*

On this page:

Ice.Override.CloseTimeout

Synopsis

`Ice.Override.CloseTimeout=`*`num`*

Description

This property overrides timeout settings used to close connections. *`num`* is the timeout value in milliseconds, or `-1` for no timeout. If this property is not defined, then `Ice.Override.Timeout` is used. If `Ice.Override.Timeout` is not defined, the endpoint timeout is used.

Ice.Override.Compress

Synopsis

`Ice.Override.Compress=`*`num`*

Description

If set, this property overrides compression settings in all proxies. If *`num`* is set to a value larger than zero, compression is enabled. If zero, compression is disabled.

The setting of this property is ignored in the server role.

Note that, if a client sets `Ice.Override.Compress=1` and sends a compressed request to a server that does not support compression, the server will close the connection and the client will receive `ConnectionLostException`.

If a client does not support compression and `Ice.Override.Compress=1`, the setting is ignored and a warning message is printed on `stderr`.

Regardless of the setting of this property, requests smaller than 100 bytes are never compressed.

Ice.Override.ConnectTimeout

Synopsis

`Ice.Override.ConnectTimeout=`*`num`*

Description

This property overrides timeout settings used to establish connections. *`num`* is the timeout value in milliseconds, or `-1` for no timeout. If this property is not defined, then `Ice.Override.Timeout` is used. If `Ice.Override.Timeout` is not defined, the endpoint timeout is used.

Ice.Override.Secure

Synopsis

`Ice.Override.Secure=`*`num`*

Description

If set to a value larger than zero, this property overrides security settings in all proxies by allowing only secure endpoints. Defining this property is equivalent to invoking the `ice_secure(true)` proxy method on every proxy. If you wish to give priority to secure endpoints without precluding the use of non-secure endpoints, use `Ice.Default.PreferSecure`. Refer to Configuring Secure Proxies for more information on secure proxies.

Ice.Override.Timeout

Synopsis

`Ice.Override.Timeout=`*`num`*

Description

If set, this property overrides timeout settings in all endpoints. *`num`* is the timeout value in milliseconds, or `-1` for no timeout.

Ice.Plugin.*

On this page:

Ice.Plugin.*name*

Synopsis

`Ice.Plugin.`*`name`*`=`*`entry_point`* `[`*`args`*`]`

Description

Defines a plug-in to be installed during communicator initialization. The format of *`entry_point`* varies by Ice implementation language, therefore this property cannot be defined in a configuration file that is shared by programs in different languages. Ice provides an alternate syntax that facilitates such sharing:

- `Ice.Plugin.`*`name`*`.clr` for the .NET Common Language Runtime
- `Ice.Plugin.`*`name`*`.cpp` for C++
- `Ice.Plugin.`*`name`*`.java` for Java

Refer to the relevant property for your language mapping for details on the entry point syntax.

Ice.Plugin.*name*.clr

Synopsis

`Ice.Plugin.`*`name`*`.clr=`*`assembly`*`:`*`class`* `[`*`args`*`]`

Description

Defines a .NET plug-in to be installed during communicator initialization. The *`assembly`* component can be a partially or fully qualified assembly name, or an assembly path name.

The details on how assemblies are load depends on how you define the *`assembly`* component and the .NET framework used by the application:

Value for *`assembly`*	Examples	Description

Assembly name	`myplugin,Version=...,Culture=neutral,publicKeyToken=...`
	
	`myplugin`
		The assembly name can be a fully or partially qualified assembly name. The assembly is loaded using Assembly.Load. For more information on how the run-time locates assemblies, see:
		• how the .NET Framework runtime locates assemblies
		• how .NET Core loads assemblies.

Assembly path name	`MyPlugin.dll`
	
	`plugins\MyPlugin.dll`
	
	`C:\plugins\MyPlugin.dll`
		The path name can be an absolute path name or a path name relative to the application's current working directory. The assembly is loaded using Assembly.LoadFrom.

The specified *`class`* must implement the `Ice.PluginFactory` interface. Any arguments that follow the class name are passed to the factory's `create` method. For example:

```
Ice.Plugin.MyPlugin.clr=MyFactory,Version=1.2.3.4:MyFactory arg1 arg2
```

Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes.

If you specify a relative path name in the entry point, the assembly is located relative to the program's current working directory:

```
Ice.Plugin.MyPlugin.clr=..\MyFactory.dll:MyFactory arg1 arg2
```

Enclose the assembly's path name in quotes if it contains spaces:

```
Ice.Plugin.MyPlugin.clr="C:\Program
Files\MyPlugin\MyFactory.dll:MyFactory" arg1 arg2
```

> A full qualified assembly name or a partial qualified assembly name in the plug-in entry point only works in .NET Core when the assembly is listed in the application dependencies file.

Ice.Plugin.*name*.cpp

Synopsis

`Ice.Plugin.`*`name`*`.cpp=`*`path`*`[,`*`version`*`]:`*`function`* `[`*`args`*`]`

Description

Defines a C++ plug-in to be installed during communicator initialization. The *`path`* and optional *`version`* components are used to construct the path name of a DLL or shared library. If no version is supplied, the Ice version is used. The *`function`* component is the name of a function with C linkage. For example, the entry point `MyPlugin,37:create` would imply a shared library name of `libMyPlugin.so.37` on Unix and `MyPlugin37.dll` on Windows. Furthermore, if Ice is built on Windows with debugging, a `d` is automatically appended to the version (for example, `MyPlugin37d.dll`). The configuration is the same for the C++11 mapping and the C++98 mapping: Ice computes the name of the shared library to load and adds automatically a "++11" suffix when needed.

The function must be declared with external linkage and have the following signature:

C++11 C++98

```
Ice::Plugin* function(const std::shared_ptr<Ice::Communicator>&
communicator,
                      const std::string& name,
                      const Ice::StringSeq& args);
```

```
Ice::Plugin* function(const Ice::CommunicatorPtr& communicator,
                      const std::string& name,
                      const Ice::StringSeq& args);
```

Note that the function must return a pointer and not a smart pointer.

Any arguments that follow the entry point are passed to the entry point function. For example:

```
Ice.Plugin.MyPlugin.cpp=MyFactory,37:create arg1 arg2
```

Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes.

The *path* component may optionally contain a relative or absolute path name, indicated by the presence of a path separator (`/` or `\`). In this case, the last component of the path is used to construct the version-specific name of the shared library or DLL. Consider this example:

```
Ice.Plugin.MyPlugin.cpp=./MyFactory,37:create arg1 arg2
```

The use of a relative path means the Ice run time will look in the current working directory for `libMyPlugin.so.37` on Unix or `MyPlugin37.dll` on Windows.

If the *path* component contains spaces, the entire entry point must be enclosed in quotes:

```
Ice.Plugin.MyPlugin.cpp="C:\Program Files\MyPlugin\MyFactory,37:create"
arg1 arg2
```

If the *path* component does not include a leading path name, Ice delegates to the operating system to locate the shared library or DLL, which typically means that the plug-in can reside in any of the directories in your shared library or DLL search path.

> When the plug-in is packaged in a static library and linked into the application through `Ice::registerPluginFactory`, the entry point (*path*[,*version*]:*function)* component of this property is ignored. The `args`, if any, are preserved, and are given to the registered plug-in factory function when the plug-in is created.

Ice.Plugin.*name*.java

Synopsis

`Ice.Plugin.`*`name`*`.java=[`*`path`*`:]`*`class`* `[`*`args`*`]`

Description

Defines a Java plug-in to be installed during communicator initialization. The specified class must implement the `com.zeroc.Ice.PluginFactory` interface. Any arguments that follow the class name are passed to the `create` method. For example:

```
Ice.Plugin.MyPlugin.java=MyFactory arg1 arg2
```

Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes.

If *path* is specified, it may be the path name of a JAR file or class directory, as shown below:

```
Ice.Plugin.MyPlugin.java=MyFactory.jar:MyFactory
Ice.Plugin.MyOtherPlugin.java=/classes:MyOtherFactory
```

If *path* contains spaces, it must be enclosed in quotes:

```
Ice.Plugin.MyPlugin.java="factory classes.jar":MyFactory
```

If *class* is specified without a path, Ice attempts to load the class using class loaders in a well-defined order.

Ice.PluginLoadOrder

Ice.PluginLoadOrder

Synopsis

`Ice.PluginLoadOrder=`*`names`*

Description

Determines the order in which plug-ins are loaded (loaded is a synonym for created in this context). The Ice run time loads the plug-ins in the order they appear in *`names`*, where each plug-in name is separated by a comma or white space. Any plug-ins not mentioned in *`names`* are loaded afterward, in an undefined order.

C++ plug-ins for which factories were registered through `Ice::registerPluginFactory` with the last parameter set to `true` are always loaded first, in registration order, and are not affected by this property.

Ice.PreferIPv6Address

Ice.PreferIPv6Address

Synopsis

`Ice.PreferIPv6Address=`*`num`*

Description

If both IPv4 and IPv6 are enabled (the default), specifies whether Ice prefers IPv6 addresses over IPv4 addresses when resolving hostnames. If *`num`* is a value greater than zero, IPv6 addresses are preferred. If not specified, the default value is 0.

Ice.TCP.*

On this page:

Ice.TCP.Backlog

Synopsis

`Ice.TCP.Backlog=`*`num`*

Description

Specifies the size of the listen queue for each TCP or SSL server endpoint. If not defined, the default value for C++ programs uses the value of `SOMAXCONN` if present, or `511` otherwise. In Java and .NET, the default value is 511.

Ice.TCP.RcvSize

Synopsis

`Ice.TCP.RcvSize=`*`num`*

Description

This property sets the TCP receive buffer size to the specified value in bytes. The default value depends on the configuration of the local TCP stack. (A common default values is 65535 bytes.)

The OS may impose lower and upper limits on the receive buffer size or otherwise adjust the buffer size. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Ice.TCP.SndSize

Synopsis

`Ice.TCP.SndSize=`*`num`*

Description

This property sets the TCP send buffer size to the specified value in bytes. The default value depends on the configuration of the local TCP stack. (A common default values is 65535 bytes.)

The OS may impose lower and upper limits on the send buffer size or otherwise adjust the buffer size. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Ice.ThreadPool.*

A communicator creates two thread pools: the client thread pool dispatches AMI callbacks and incoming requests on bidirectional connections, and the server thread pool dispatches requests to object adapters.

This page describes configuration properties for the client and server thread pools. These thread pools are named `Client` and `Server`, respectively. In the property descriptions below, replace *name* with `Client` or `Server`.

On this page:

Ice.ThreadPool.*name*.Serialize

Synopsis

`Ice.ThreadPool.`*`name`*`.Serialize=`*`num`*

Description

If *num* is a value greater than 0, the `Client` or `Server` thread pool serializes all messages from each connection. It is not necessary to enable this feature in a thread pool whose maximum size is 1 thread. When a thread pool dispatches requests implemented with AMD, it serializes the dispatching of requests from each connection, but it does not wait for a request to complete before it dispatches the next request.

In a multi-threaded pool, enabling serialization allows requests from different connections to be dispatched concurrently while preserving the order of messages on each connection. Note that serialization can have a significant impact on latency and throughput. If not defined, the default value is 0.

Ice.ThreadPool.*name*.Size

Synopsis

`Ice.ThreadPool.`*`name`*`.Size=`*`num`*

Description

Thread pools in Ice can grow and shrink dynamically, based on an average load factor. A thread pool always has at least 1 thread and may grow as load increases up to the maximum size specified by `Ice.ThreadPool.`*`name`*`.SizeMax`. If `SizeMax` is not specified, Ice uses the value of *num* as the pool's maximum size. The `Client` or `Server` thread pool is initialized with *num* active threads, but the pool may shrink to only 1 thread during idle periods as determined by `Ice.ThreadPool.`*`name`*`.ThreadIdleTime`.

If not specified, the default value is 1 for both properties.

An object adapter can also be configured with its own thread pool.

Note that multiple threads for the client thread pool are only required for nested AMI invocations, or to allow multiple AMI callbacks to be dispatched concurrently.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.*name*.SizeMax

Synopsis

`Ice.ThreadPool.`*`name`*`.SizeMax=`*`num`*

Description

`num` is the maximum number of threads for the `Client` or `Server` thread pool. Refer to the `Ice.ThreadPool.`*`name`*`.Size` property for more information on configuring the size of a thread pool.

The default value for `SizeMax` is the value of `Size`, meaning the thread pool can never grow larger than its initial size.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.*name*.SizeWarn

Synopsis

`Ice.ThreadPool.`*`name`*`.SizeWarn=`*`num`*

Description

Whenever *`num`* threads are active in the `Client` or `Server` thread pool, a "low on threads" warning is printed. The default value is 0, which disables the warning.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.*name*.StackSize

Synopsis

`Ice.ThreadPool.`*`name`*`.StackSize=`*`num`*

Description

`num` is the stack size (in bytes) of threads in the `Client` or `Server` thread pool. The default value is 0, meaning the operating system's default is used.

Ice.ThreadPool.*name*.ThreadIdleTime

Synopsis

`Ice.ThreadPool.`*`name`*`.ThreadIdleTime=`*`num`*

Description

Ice can automatically reap idle threads in the `Client` or `Server` thread pool to conserve resources. This property specifies the number of seconds a thread must be idle before it is reaped. If not specified, the default value is 60 seconds.

> The threads in Ice thread pools are assigned jobs at random, and this randomness affects how quickly a thread in an under-utilized thread pool will get reaped.

To disable the reaping of idle threads, set `ThreadIdleTime` to 0. In this situation, the thread pool is initialized with `Ice.ThreadPool.`*`name`*`.Size` active threads and may grow to contain `Ice.ThreadPool.`*`name`*`.SizeMax` active threads, but the size of the pool never decreases.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.*name*.ThreadPriority

Synopsis

`Ice.ThreadPool.`*`name`*`.ThreadPriority=`*`num`*

Description

`num` specifies a thread priority for the threads in the `Client` or `Server` thread pool. Leaving this property unset causes the run time to create threads with the default priority specified by `Ice.ThreadPriority`.

This property is unset by default.

You can also override the default priority for a specific object adapter using *`adapter`*`.ThreadPool.ThreadPriority`.

Ice.ThreadPriority

Ice.ThreadPriority

Synopsis

`Ice.ThreadPriority=`*`num`*

Description

`num` specifies a thread priority. Threads created by the Ice run time are created with the specified priority by default. Leaving this property unset causes the run time to create threads with the system default priority. This property is unset by default.

You can separately override the default priorities for the client and server thread pools using `Ice.ThreadPool.`*`name`*`.ThreadPriority` a s well as for a specific object adapter using *`adapter`*`.ThreadPool.ThreadPriority`.

Ice.Trace.*

On this page:

Ice.Trace.Admin.Logger

Synopsis

`Ice.Trace.Admin.Logger=`*`num`*

Description

Controls the trace level for the Logger administrative facet.

0	No trace (default).
1	Trace when a remote logger is attached or detached.
2	Like 1, but also trace the sending of log messages to remote loggers.

Ice.Trace.Admin.Properties

Synopsis

`Ice.Trace.Admin.Properties=`*`num`*

Description

Controls the trace level for property updates made via the `Properties` facet:

0	No property trace (default).
1	Trace property addition, modification, and removal.

Ice.Trace.Locator

Synopsis

`Ice.Trace.Locator=`*`num`*

Description

The Ice run time makes locator requests to resolve the endpoints of object adapters and well-known objects. Requests on the locator registry are used to update object adapter endpoints and set the server process proxy. This property controls the trace level for the Ice run time's interactions with the locator:

0	No locator trace (default).
1	Trace Ice locator and locator registry requests.
2	Like 1, but also trace the removal of endpoints from the cache.

Ice.Trace.Network

Synopsis

`Ice.Trace.Network=`*`num`*

Description

Controls the trace level for low-level network activities such as connection establishment and read/write operations:

0	No network trace (default).
1	Trace successful connection establishment and closure.
2	Like 1, but also trace attempts to bind, connect, and disconnect sockets as well as Ice endpoint usage.
3	Like 2, but also trace data transfer, the published endpoints for an object adapter, and the current list of local addresses for an endpoint that uses the wildcard address.

Ice.Trace.Protocol

Synopsis

`Ice.Trace.Protocol=`*`num`*

Description

Controls the trace level for Ice protocol mesages:

0	No protocol trace (default).
1	Trace Ice protocol messages.

Ice.Trace.Retry

Synopsis

`Ice.Trace.Retry=`*`num`*

Description

Ice supports automatic retries in case of a request failure. This property controls the trace level for retry attempts:

0	No request retry trace (default).
1	Trace Ice operation call retries.
2	Also trace Ice retry for connection establishment failures on Ice locator cached endpoints.

Ice.Trace.Slicing

Synopsis

`Ice.Trace.Slicing=`*`num`*

Description

The Ice data encoding for exceptions and classes enables a receiver to slice an unknown exception or class type to a known type. This

property controls the trace level for slicing activities:

0	No trace of slicing activity (default).
1	Trace all exception and class types that are unknown to the receiver and therefore sliced.

Ice.Trace.ThreadPool

Synopsis

`Ice.Trace.ThreadPool=`*`num`*

Description

Controls the trace level for the Ice thread pool:

0	No trace of thread pool activity (default).
1	Trace the creation, growing, and shrinking of thread pools.

Ice.UDP.*

On this page:

Ice.UDP.RcvSize

Synopsis

`Ice.UDP.RcvSize=`*`num`*

Description

This property sets the UDP receive buffer size to the specified value in bytes. Ice messages larger than *`num`* `- 28` bytes cause a `DatagramLimitException`. The default value depends on the configuration of the local UDP stack. (Common default values are 65535 and 8192 bytes.)

The OS may impose lower and upper limits on the receive buffer size or otherwise adjust the buffer size. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Values less than 28 are ignored.

Note that, on many operating systems, it is possible to set a buffer size greater than 65535. Such settings do not change the hard limit of 65507 bytes for the payload of a UDP packet, but merely affect how much data can be buffered by the kernel.

Settings less than 65535 limit the size of Ice datagrams as well as adjust the kernel buffer sizes.

Ice.UDP.SndSize

Synopsis

`Ice.UDP.SndSize=`*`num`*

Description

This property sets the UDP send buffer size to the specified value in bytes. Ice messages larger than *`num`* `- 28` bytes cause a `DatagramLimitException`. The default value depends on the configuration of the local UDP stack. (Common default values are 65535 and 8192 bytes.)

The OS may impose lower and upper limits on the send buffer size or otherwise adjust the buffer size. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Values less than 28 are ignored.

Note that, on many operating systems, it is possible to set a buffer size greater than 65535. Such settings do not change the hard limit of 65507 bytes for the payload of a UDP packet, but merely affect how much data can be buffered by the kernel.

Settings less than 65535 limit the size of Ice datagrams as well as adjust the kernel buffer sizes.

Ice.Warn.*

On this page:

Ice.Warn.AMICallback

Synopsis

`Ice.Warn.AMICallback=`*`num`*

Description

If *`num`* is set to a value larger than zero, warnings are printed if an AMI callback raises an exception. The default value is 1.

Ice.Warn.Connections

Synopsis

`Ice.Warn.Connections=`*`num`*

Description

If *`num`* is set to a value larger than zero, Ice applications print warning messages for certain exceptional conditions in connections. The default value is 0.

Ice.Warn.Datagrams

Synopsis

`Ice.Warn.Datagrams=`*`num`*

Description

If *`num`* is set to a value larger than zero, a server prints a warning message if it receives a datagram that exceeds the server's receive buffer size. (Note that this condition is not detected by all UDP implementations — some implementations silently drop received datagrams that are too large.) The default value is 0.

Ice.Warn.Dispatch

Synopsis

`Ice.Warn.Dispatch=`*`num`*

Description

If *`num`* is set to a value larger than zero, Ice applications print warning messages for certain exceptions that are raised while an incoming request is dispatched.

Warning levels:

0	No warnings.

1	Print warnings for unexpected `Ice::LocalException`, `Ice::UserException`, C++ exceptions, and Java run-time exceptions (default).
2	Like 1, but also issue warnings for `Ice::ObjectNotExistException`, `Ice::FacetNotExistException`, and `Ice::OperationNotExistException`.

Ice.Warn.Endpoints

Synopsis

`Ice.Warn.Endpoints=`*`num`*

Description

If *`num`* is set to a value larger than zero, a warning is printed if a stringified proxy contains an endpoint that cannot be parsed. (For example, stringified proxies containing SSL endpoints cause this warning in versions of Ice that do not support SSL.) The default value is 1.

Ice.Warn.UnknownProperties

Synopsis

`Ice.Warn.UnknownProperties=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Ice run time prints a warning about unknown properties for object adapters and proxies. The default value is 1.

Ice.Warn.UnusedProperties

Synopsis

`Ice.Warn.UnusedProperties=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Ice run time prints a warning during communicator destruction about properties that were set but not read. This warning is useful for detecting mis-spelled properties, such as `Filesystem.MaxFilSize`. The default value is 0.

IceBox.*

On this page:

IceBox.InheritProperties

Synopsis

`IceBox.InheritProperties=`*`num`*

Description

If *`num`* is set to a value larger than zero, each service inherits the configuration properties of the IceBox server's communicator. If not defined, the default value is zero.

IceBox.LoadOrder

Synopsis

`IceBox.LoadOrder=`*`names`*

Description

Determines the order in which services are loaded. The service manager loads the services in the order they appear in *`names`*, where each service name is separated by a comma or white space. Any services not mentioned in *`names`* are loaded afterward, in an undefined order.

IceBox.PrintServicesReady

Synopsis

`IceBox.PrintServicesReady=`*`token`*

Description

If this property is set to a value greater than zero, the service manager prints "*`token`* ready" on standard output once initialization of all the services is complete. This is useful for scripts that need to wait until all services are ready to be used.

IceBox.Service.*name*

Synopsis

`IceBox.Service.`*`name=entry_point`* `[`*`args`*`]`

Description

Defines a service to be loaded during IceBox initialization. Any arguments that follow the entry point are examined; those matching the `--`*`name`*`.*=`*`value`* pattern are interpreted as property definitions and appear in the property set of the communicator that is passed to the service `start` method, and all remaining arguments are passed to the `start` method in the `args` parameter. Whitespace separates the arguments, and any arguments that contain whitespace must be enclosed in quotes.

Platform Notes

C++

The value of *entry_point* has the following form:

path[,*version*]:*function*

The *path* and optional *version* components are used to construct the name of a DLL or shared library. If no version is supplied, the version is the empty string. The *function* component is the name of a function with extern C linkage. For example, the entry point `IceStormService,37:createIceStorm` implies a shared library name of `libIceStormService.so.37` on Unix and `IceStormService37.dll` on Windows. Furthermore, if IceBox is built on Windows with debugging, a `d` is automatically appended to the version (e.g., `IceStormService37d.dll`). The configuration is the same for the C++11 mapping and the C++98 mapping: Ice computes the name of the shared library to load and adds automatically a "++11" suffix when needed.

The function must be declared with extern C linkage and have the following signature:

C++11 C++98

```
IceBox::Service* function(const std::shared_ptr<Ice::Communicator>& c);
```

```
IceBox::Service* function(Ice::CommunicatorPtr c);
```

Note that the function must return a pointer and not a smart pointer. IceBox deallocates the object when it unloads the library. The communicator instance passed to this function is the server's communicator, which is not the same as the communicator passed to the service's `start` method.

The *path* component may optionally contain a relative or absolute path name, indicated by the presence of a path separator (`/` or `\`). In this case, the last component of the path is used to construct the name of the shared library or DLL. Consider this example:

```
IceBox.Service.IceStorm=./IceStormService,37:createIceStorm
```

The use of a relative path means the Ice run time will look in the current working directory for `libIceStormService.so.37` on Unix or `IceStormService37.dll` on Windows.

If the *path* component contains spaces, the entire entry point must be enclosed in quotes:

```
IceBox.Service.IceStorm="C:\Program
Files\ZeroC\Ice-3.7\bin\IceStormService,37:createIceStorm"
```

If the *path* component does not include a leading path name, Ice delegates to the operating system to locate the shared library or DLL, which typically means that the plug-in can reside in any of the directories in your shared library or DLL search path.

Java

The value of *entry_point* has the following form:

[*path*:]*class*

The *class* component must be the name of a class that implements the `com.zeroc.IceBox.Service` interface and provides at least one of the constructors shown in the example below:

Java

```
public class MyService implements com.zeroc.IceBox.Service
{
    public MyService(com.zeroc.Ice.Communicator serverCommunicator);
    public MyService();

    // ...
}
```

The constructor taking a `Communicator` argument is invoked if present, otherwise the default constructor is invoked.

If *`path`* is specified, it may be the path name of a JAR file or class directory, as shown below:

```
IceBox.Service.MyService=MyService.jar:MyServiceImpl
IceBox.Service.MyOtherService=/classes:MyOtherServiceImpl
```

If *`path`* contains spaces, it must be enclosed in quotes:

```
IceBox.Service.MyService="factory classes.jar":MyServiceImpl
```

IceBox uses a single class loader to load all services having the same value for *`path`*.

If *`class`* is specified without a path, IceBox attempts to load the class using class loaders in a well-defined order.

.NET

The value of *`entry_point`* has the form *`assembly`*`:`*`class`*. The *`assembly`* can be a partially or fully qualified assembly name, such as `myplugin,Version=0.0.0.0,Culture=neutral`, or an assembly DLL name such as `myplugin.dll`, and may optionally include a leading relative or absolute path name.

> You *must* use a fully-qualified assembly name to load a service from an assembly in the Global Assembly Cache.

The specified class must implement the `IceBox.Service` interface and provide at least one of the constructors shown in the example below:

C#

```
public class MyService : IceBox.Service
{
    public MyService(Ice.Communicator serverCommunicator);
    public MyService();

    // ...
}
```

The constructor taking an `Ice.Communicator` argument is invoked if present, otherwise the default constructor is invoked.

If you specify a relative path name in the entry point, the assembly is located relative to the program's current working directory:

```
IceBox.Service.MyService=..\MyService.dll:MyServiceImpl
```

Enclose the assembly's path name in quotes if it contains spaces:

```
IceBox.Service.MyService="C:\Program
Files\MyService\MyService.dll:MyServiceImpl"
```

Finally, if the assembly uses a leading path name, be sure to include the `.dll` extension.

IceBox.UseSharedCommunicator.*name*

Synopsis

`IceBox.UseSharedCommunicator.`*`name`*`=`*`num`*

Description

If *`num`* is set to a value larger than zero, the service manager supplies the service *`name`* with a communicator that might be shared by other services. If the `IceBox.InheritProperties` property is also defined, the shared communicator inherits the properties of the IceBox server. If not defined, the default value is zero.

IceBoxAdmin

IceBoxAdmin.ServiceManager.Proxy

Synopsis

`IceBoxAdmin.ServiceManager.Proxy=`*`proxy`*

Description

This property configures the proxy that is used by the `iceboxadmin` utility to locate the service manager.

IceBridge.*

IceBridge is an Ice service that acts as a bridge between one or more clients and a server.

On this page:

IceBridge.InstanceName

Synopsis

`IceBridge.InstanceName=`*`name`*

Description

Specifies a default identity category for IceBridge objects. If defined, the identity of the IceBridge router interface becomes *`name`*`/router`.

If not defined, the default value is `IceBridge`.

IceBridge.Source.*AdapterProperty*

Synopsis

`IceBridge.Source.`*`AdapterProperty=value`*

Description

IceBridge uses the adapter name `IceBridge.Source` for the object adapter that it provides to clients. Therefore, adapter properties can be used to configure this adapter. The only required adapter property is `IceBridge.Source.Endpoints`.

This adapter must be accessible to IceBridge clients.

IceBridge.Target.Endpoints

Synopsis

`IceBridge.Target.Endpoints=`*`endpoints`*

Description

This property specifies the endpoints of the target server. For each new connection that a client establishes to an endpoint in `IceBridge.Source.Endpoints`, IceBridge will create a matching outgoing connection to a target endpoint.

IceBT.*

IceBT is the Bluetooth transport plug-in for Android and Linux.

On this page:

IceBT.RcvSize

Synopsis

`IceBT.RcvSize=`*`num`*

Description

This property sets the receive buffer size to the specified value in bytes.

Platform Notes

Linux

The default value depends on the configuration of the local Bluetooth stack.

The OS may impose lower and upper limits on the receive buffer size or otherwise adjust the buffer size. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

IceBT.SndSize

Synopsis

`IceBT.SndSize=`*`num`*

Description

This property sets the send buffer size to the specified value in bytes.

Platform Notes

Linux

The default value depends on the configuration of the local Bluetooth stack.

The OS may impose lower and upper limits on the send buffer size or otherwise adjust the buffer size. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

IceDiscovery.*

This page describes the properties supported by the IceDiscovery plug-in.

On this page:

IceDiscovery.Address

Synopsis

`IceDiscovery.Address=`*`addr`*

Description

Specifies the multicast IP address to use for sending or receiving multicast discovery queries. If not defined, the default value depends on other property settings:

- If `Ice.PreferIPv6Address` is enabled or `Ice.IPv4` is disabled, IceDiscovery uses the IPv6 address `ff15::1`
- Otherwise IceDiscovery uses `239.255.0.1`

This property is used to compose the value of `IceDiscovery.Lookup` and `IceDiscovery.Multicast.Endpoints`.

IceDiscovery.DomainId

Synopsis

`IceDiscovery.DomainId=`*`id`*

Description

Specifies the domain ID used to locate objects and object adapters. The IceDiscovery plug-in only responds to requests from clients with the same domain ID and ignores requests from clients with a different domain ID. If not defined, the default domain ID is an empty string.

IceDiscovery.Interface

Synopsis

`IceDiscovery.Interface=`*`intf`*

Description

Specifies the IP address of the interface to use for sending or receiving multicast discovery queries. If not defined, the discovery will use all the network interfaces available on the system to send and receive UDP multicast datagrams. This property is used to compose the value of `IceDiscovery.Lookup`, `IceDiscovery.Reply.Enpoints` and `IceDiscovery.Multicast.Endpoints`.

IceDiscovery.Lookup

Synopsis

`IceDiscovery.Lookup=`*`endpoints`*

Description

Specifies the multicast endpoints that a client uses to send multicast discovery queries. If not defined, the endpoint is composed as follows:

```
udp -h addr -p port [--interface intf]
```

where *`addr`* is the value of `IceDiscovery.Address`, *`port`* is the value of `IceDiscovery.Port`, and *`intf`* is the value of `IceDiscovery.Interface`. If multiple endpoints are defined, the queries will be sent on each endpoint.

IceDiscovery.Multicast.*AdapterProperty*

Synopsis

`IceDiscovery.Multicast.`*`AdapterProperty=value`*

Description

IceDiscovery creates an object adapter named `IceDiscovery.Multicast` for receiving discovery queries from clients. If not otherwise defined by `IceDiscovery.Multicast.Endpoints`, the endpoint for this object adapter is composed as follows:

```
udp -h addr -p port [--interface intf]
```

where *`addr`* is the value of `IceDiscovery.Address`, *`port`* is the value of `IceDiscovery.Port`, and *`intf`* is the value of `IceDiscovery.Interface`.

You don't normally need to set other properties for this object adapter.

IceDiscovery.LatencyMultiplier

Synopsis

`IceDiscovery.LatencyMultiplier=`*`num`*

Description

Specifies the multiplier to apply to the latency of the first discovery request-reply for replica groups. When IceDiscovery receives a reply for a discovery request and this reply indicates that the adapter identifier is a replica group, it waits for an additional time interval for other responses from replicated servers. This time interval is based on the latency of the first request-reply and the latency multiplier. For example, if the first reply is received after 15 milliseconds and the multiplier is set to 4, IceDiscovery will wait for an additional 60 milliseconds for replies from other servers. If not defined, the default is `1`.

IceDiscovery.Port

Synopsis

`IceDiscovery.Port=`*`port`*

Description

Specifies the multicast port to use for sending or receiving multicast requests. If not set, the default value is `4061`.

IceDiscovery.Reply.*AdapterProperty*

Synopsis

`IceDiscovery.Reply.`*`AdapterProperty=value`*

Description

IceDiscovery creates an object adapter named `IceDiscovery.Reply` for receiving replies to multicast requests. If not otherwise defined by `IceDiscovery.Reply.Endpoints`, the endpoint for this object adapter is composed as follows:

```
udp [-h intf]
```

where *`intf`* is the value of `IceDiscovery.Interface`. A fixed port is not necessary for this endpoint.

You don't normally need to set other properties for this object adapter.

IceDiscovery.RetryCount

Synopsis

`IceDiscovery.RetryCount=`*`num`*

Description

Specifies the maximum number of times that the plug-in will retry sending UDP multicast requests before giving up. The `IceDiscovery.Timeout` property determines how long the plug-in waits for a reply before trying again. If not defined, the default retry count is 3.

IceDiscovery.Timeout

Synopsis

`IceDiscovery.Timeout=`*`num`*

Description

Specifies the time interval in milliseconds to wait for replies to UDP multicast requests. If no server replies during this time interval, the client will retry the request the number of times specified by `IceDiscovery.RetryCount`. If not defined, the default timeout is `300`.

IceGrid.*

On this page:

IceGrid.InstanceName

Synopsis

`IceGrid.InstanceName=`*`name`*

Description

Specifies an alternate identity category for the well-known IceGrid objects. If defined, the identities of the IceGrid objects become:

`name``/AdminSessionManager`
`name``/AdminSessionManager-replica`
`name``/AdminSSLSessionManager`
`name``/AdminSSLSessionManager-replica`
`name``/NullPermissionsVerifier`
`name``/NullSSLPermissionsVerifier`
`name``/Locator`
`name``/Query`
`name``/Registry`
`name``/Registry-replica`
`name``/RegistryUserAccountMapper`
`name``/RegistryUserAccountMapper-replica`
`name``/SessionManager`
`name``/SSLSessionManager`

If not specified, the default identity category is `IceGrid`.

IceGrid.Node.*AdapterProperty*

Synopsis

`IceGrid.Node.`*`AdapterProperty`*`=`*`value`*

Description

An IceGrid node uses the adapter name `IceGrid.Node` for the object adapter that the registry contacts to communicate with the node. Therefore, adapter properties can be used to configure this adapter.

IceGrid.Node.AllowEndpointsOverride

Synopsis

`IceGrid.Node.AllowEndpointsOverride=`*`num`*

If *`num`* is set to a non-zero value, an IceGrid node permits servers to override previously set endpoints even if the server is active. Setting this property to a non-zero value is necessary if the servers managed by the node use the object adapter operation `refreshPublishedEndpoints`. The default value of *`num`* is zero.

IceGrid.Node.AllowRunningServersAsRoot

Synopsis

`IceGrid.Node.AllowRunningServersAsRoot=`*`num`*

If *`num`* is set to a non-zero value, an IceGrid node will permit servers started by the node to run with super-user privileges. Note that you should not set this property unless the node uses a secure endpoint; otherwise, clients can start arbitrary processes with super-user privileges on the node's machine.

The default value of *`num`* is zero.

IceGrid.Node.CollocateRegistry

Synopsis

`IceGrid.Node.CollocateRegistry=`*`num`*

Description

If *`num`* is set to a value larger than zero, the node collocates the IceGrid registry.

The collocated registry is configured with the same properties as the standalone IceGrid registry.

IceGrid.Node.Data

Synopsis

`IceGrid.Node.Data=`*`path`*

Description

Defines the path of the IceGrid node data directory. The node creates `distrib`, `servers`, and `tmp` subdirectories in this directory if they do not already exist. The `distrib` directory contains distribution files downloaded by the node from an IcePatch2 server. The `servers` directory contains configuration data for each deployed server. The `tmp` directory holds temporary files.

IceGrid.Node.DisableOnFailure

Synopsis

`IceGrid.Node.DisableOnFailure=`*`num`*

Description

The node considers a server to have terminated improperly if it has a non-zero exit code or if it exits due to one of the signals `SIGABRT`, `SIGBUS`, `SIGILL`, `SIGFPE`, or `SIGSEGV`. The node marks such a server as disabled if *`num`* is a non-zero value; a disabled server cannot be activated on demand. For values of *`num`* greater than zero, the server is disabled for *`num`* seconds. If *`num`* is a negative value, the server is disabled indefinitely, or until it is explicitly enabled or started via an administrative action. The default value is zero, meaning the node does not disable servers in this situation.

IceGrid.Node.Name

Synopsis

`IceGrid.Node.Name=`*`name`*

Description

Defines the *`name`* of the IceGrid node. All nodes using the same registry must have unique names; a node refuses to start if there is a node with the same name running already. This property must be defined for each node.

IceGrid.Node.Output

Synopsis

`IceGrid.Node.Output=`*`path`*

Description

Defines the path of the IceGrid node output directory. If set, the node redirects the `stdout` and `stderr` output of the started servers to files named `server.out` and `server.err` in this directory. Otherwise, the started servers share the `stdout` and `stderr` of the node's process.

IceGrid.Node.PrintServersReady

Synopsis

`IceGrid.Node.PrintServersReady=`*`token`*

Description

The IceGrid node prints "*`token`* ready" on standard output after all the servers managed by the node are ready. This is useful for scripts that wish to wait until all servers have been started and are ready for use.

IceGrid.Node.ProcessorSocketCount

Synopsis

`IceGrid.Node.ProcessorSocketCount=`*`num`*

Description

This property sets the number of processor sockets. This value is reported by the `icegridadmin` `node sockets` command. On Windows Vista (or later), Windows Server 2008 (or later), and Linux systems, the number of processor sockets is set automatically by the Ice run time. On other systems, the run time cannot obtain the socket count from the operating system; you can use this property to set the number of processor sockets manually on such systems.

IceGrid.Node.PropertiesOverride

Synopsis

`IceGrid.Node.PropertiesOverride=`*`overrides`*

Description

Defines a list of properties that override the properties defined in server deployment descriptors. For example, in some cases it is desirable to set the property `Ice.Default.Host` for servers, but not in server deployment descriptors. The property definitions must be separated by white space.

IceGrid.Node.RedirectErrToOut

Synopsis

`IceGrid.Node.RedirectErrToOut=`*`num`*

Description

If *`num`* is set to a value larger than zero, the `stderr` of each started server is redirected to the server's `stdout`.

IceGrid.Node.Trace.Activator

Synopsis

`IceGrid.Node.Trace.Activator=`*`num`*

Description

The activator trace level:

0	No activator trace (default).
1	Trace process activation, termination.
2	Like 1, but more verbose, including process signaling and more diagnostic messages on process activation.
3	Like 2, but more verbose, including more diagnostic messages on process activation (e.g., path, working directory, and arguments of the activated process).

IceGrid.Node.Trace.Adapter

Synopsis

`IceGrid.Node.Trace.Adapter=`*`num`*

Description

The object adapter trace level:

0	No object adapter trace (default).
1	Trace object adapter addition, removal.
2	Like 1, but more verbose, including object adapter activation and deactivation and more diagnostic messages.
3	Like 2, but more verbose, including object adapter transitional state change (for example, "waiting for activation").

IceGrid.Node.Trace.Admin

Synopsis

`IceGrid.Node.Trace.Admin=`*`num`*

Description

Set the trace level for the routing of operations to Ice.Admin objects through this node.

0	No admin trace (default).
1	Trace routing of operations to Ice.Admin objects

IceGrid.Node.Trace.Patch

Synopsis

`IceGrid.Node.Trace.Patch=`*`num`*

Description

The patch trace level:

0	No patching trace (default).
1	Show summary of patch progress.
2	Like 1, but more verbose, including download statistics.
3	Like 2, but more verbose, including checksum information.

IceGrid.Node.Trace.Replica

Synopsis

`IceGrid.Node.Trace.Replica=`*`num`*

Description

The replica trace level:

0	No replica trace (default).
1	Trace session lifecycle between nodes and replicas.
2	Like 1, but more verbose, including session establishment attempts and failures.
3	Like 2, but more verbose, including keep alive messages sent to the replica.

IceGrid.Node.Trace.Server

Synopsis

`IceGrid.Node.Trace.Server=`*`num`*

Description

The server trace level:

0	No server trace (default).
1	Trace server addition, removal.
2	Like 1, but more verbose, including server activation and deactivation, property updates, and more diagnostic messages.
3	Like 2, but more verbose, including server transitional state change (activating and deactivating).

IceGrid.Node.UserAccountMapper

Synopsis

`IceGrid.Node.UserAccountMapper=`*`proxy`*

Description

Specifies the proxy of an object that implements the `IceGrid::UserAccountMapper` interface for customizing the user accounts under which servers are activated. The IceGrid node invokes this proxy to map session identifiers (the user ID for sessions created with a user ID and password, or the distinguished name for sessions created from a secure connection) to user accounts.

As a proxy property, you can configure additional aspects of the proxy using properties.

IceGrid.Node.UserAccounts

Synopsis

`IceGrid.Node.UserAccounts=`*`file`*

Description

Specifies the file name of an IceGrid node user account map file. Each line of the file must contain an identifier and a user account, separated by white space. The identifier will be matched against the client session identifier (the user ID for sessions created with a user ID

and password, or the distinguished name for sessions created from a secure connection). This user account map file is used by the node to map session identifiers to user accounts. This property is ignored if `IceGrid.Node.UserAccountMapper` is defined.

IceGrid.Node.WaitTime

Synopsis

`IceGrid.Node.WaitTime=`*`num`*

Description

Defines the interval in seconds that IceGrid waits for server activation and deactivation.

If a server is automatically activated and does not register its object adapter endpoints within this time interval, the node assumes there is a problem with the server and returns an empty set of endpoints to the client.

If a server is being gracefully deactivated and IceGrid does not detect the server deactivation during this time interval, IceGrid kills the server.

The default value is 60 seconds.

IceGrid.Registry.AdminCryptPasswords

Synopsis

`IceGrid.Registry.AdminCryptPasswords=`*`file`*

Description

Specifies the file name of an IceGrid registry access control list for administrative clients. Each line of the file must contain a user name and a password, separated by white space. The password must a `MCF` encoded string as described here. If this property is not defined, the default value is `admin-passwords`. This property is ignored if `IceGrid.Registry.AdminPermissionsVerifier` is defined.

IceGrid.Registry.AdminPermissionsVerifier

Synopsis

`IceGrid.Registry.AdminPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::PermissionsVerifier` interface for controlling access to IceGrid administrative sessions. The IceGrid registry invokes this proxy to validate each new administrative session created by a client with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using properties.

IceGrid.Registry.AdminSessionFilters

Synopsis

`IceGrid.Registry.AdminSessionFilters=`*`num`*

Description

This property controls whether IceGrid establishes filters for sessions created with the IceGrid session manager. If *`num`* is set to a value larger than zero, IceGrid establishes these filters, so Glacier2 limits access to the `IceGrid::AdminSession` object and the `IceGrid::Admin` object that is returned by the `getAdmin` operation. If *`num`* is set to zero, IceGrid does not establish filters, so access to these objects is controlled solely by Glacier2's configuration.

The default value is 1.

IceGrid.Registry.AdminSessionManager.*AdapterProperty*

Synopsis

`IceGrid.Registry.AdminSessionManager.`*`AdapterProperty`*`=`*`value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.AdminSessionManager` for the object adapter that processes incoming requests from IceGrid administrative sessions. Therefore, adapter properties can be used to configure this adapter. (Note any setting of `IceGrid.Registry.AdminSessionManager.AdapterId` is ignored because the registry always provides a direct adapter.)

For security reasons, defining endpoints for this object adapter is optional. If you do define endpoints, they should only be accessible to Glacier2 routers used to create IceGrid administrative sessions.

IceGrid.Registry.AdminSSLPermissionsVerifier

Synopsis

`IceGrid.Registry.AdminSSLPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLPermissionsVerifier` interface for controlling access to IceGrid administrative sessions. The IceGrid registry invokes this proxy to validate each new administrative session created by a client from a secure connection with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using the properties.

IceGrid.Registry.Client.*AdapterProperty*

Synopsis

`IceGrid.Registry.Client.`*`AdapterProperty`*`=`*`value`*

Description

IceGrid uses the adapter name `IceGrid.Registry.Client` for the object adapter that processes incoming requests from clients. Therefore, adapter properties can be used to configure this adapter. (Note any setting of `IceGrid.Registry.Client.AdapterId` is ignored because the registry always provides a direct adapter.)

Note that `IceGrid.Registry.Client.Endpoints` controls the client endpoint for the registry. The port numbers 4061 (for TCP) and 4062 (for SSL) are reserved for the registry by the Internet Assigned Numbers Authority (IANA).

IceGrid.Registry.CryptPasswords

Synopsis

`IceGrid.Registry.CryptPasswords=`*`file`*

Description

Specifies the file name of an IceGrid registry access control list. Each line of the file must contain a user name and a password, separated by white space. The password must a `MCF` encoded string as described here. If this property is not defined, the default value is `passwords`. This property is ignored if `IceGrid.Registry.PermissionsVerifier` is defined.

IceGrid.Registry.DefaultTemplates

Synopsis

`IceGrid.Registry.DefaultTemplates=`*`path`*

Description

Defines the path name of an XML file containing default template descriptors. A sample file named `config/templates.xml` that contains convenient server templates for Ice services is provided in the Ice distribution.

IceGrid.Registry.Discovery.*AdapterProperty*

Synopsis

`IceGrid.Registry.Discovery.`*`AdapterProperty`*`=`*`value`*

Description

The IceGrid registry creates an object adapter named `IceGrid.Registry.Discovery` for receiving multicast discovery queries from clients. If not otherwise defined by `IceGrid.Registry.Discovery.Endpoints`, the endpoint for this object adapter is composed as follows:

```
udp -h addr -p port [--interface intf]
```

where *`addr`* is the value of `IceGrid.Registry.Discovery.Address`, *`port`* is the value of `IceGrid.Registry.Discovery.Port`, and *`intf`* is the value of `IceGrid.Registry.Discovery.Interface`.

You don't normally need to set other properties for this object adapter.

IceGrid.Registry.Discovery.Address

Synopsis

`IceGrid.Registry.Discovery.Address=`*`addr`*

Description

Specifies the multicast IP address to use for receiving multicast discovery queries. If not defined, the default value depends on the setting of `Ice.IPv4`: if enabled (the default), IceDiscovery uses the address `239.255.0.1`, otherwise IceDiscovery assumes the application wants to use IPv6 and defaults to the address `ff15::1` instead. This property is used to compose the endpoint of the `IceGrid.Registry.Discovery` object adapter.

IceGrid.Registry.Discovery.Enabled

Synopsis

`IceDiscovery.Enabled=`*`num`*

Description

If *`num`* is a value larger than zero, the registry creates the `IceGrid.Registry.Discovery` object adapter and listens for multicast discovery queries. If not defined, the default value is `1`. Set this property to zero to disable multicast discovery.

IceGrid.Registry.Discovery.Interface

Synopsis

`IceGrid.Registry.Discovery.Interface=`*`intf`*

Description

Specifies the IP address of the interface to use for receiving multicast discovery queries. If not defined, the operating system will select a default interface to send and receive the UDP multicast datagrams. This property is used to compose the endpoint of the `IceGrid.Registry.Discovery` object adapter.

IceGrid.Registry.Discovery.Port

Synopsis

`IceGrid.Registry.Discovery.Port=`*`port`*

Description

Specifies the multicast port to use for receiving multicast discovery queries. If not set, the default value is `4061`. This property is used to compose the endpoint of the `IceGrid.Registry.Discovery` object adapter.

IceGrid.Registry.DynamicRegistration

Synopsis

`IceGrid.Registry.DynamicRegistration=`*`num`*

Description

If *`num`* is set to a value larger than zero, the locator registry does not require Ice servers to preregister object adapters and replica groups, but rather creates them automatically if they do not exist. If this property is not defined, or *`num`* is set to zero, an attempt to register an unknown object adapter or replica group causes adapter activation to fail with `Ice.NotRegisteredException`. An object adapter registers itself when the *`adapter`*`.AdapterId` property is defined. The *`adapter`*`.ReplicaGroupId` property identifies the replica group. An adapter registered with dynamic registration can only be a member of a replica group also registered with dynamic registration. Trying to dynamically register an adapter with a replica group registered with the deployment facility will fail with `Ice.NotRegisteredException`.

IceGrid.Registry.Internal.*AdapterProperty*

Synopsis

`IceGrid.Registry.Internal.`*`AdapterProperty=value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.Internal` for the object adapter that processes incoming requests from nodes and slave replicas. Therefore, adapter properties can be used to configure this adapter. (Note any setting of `IceGrid.Registry.Internal.AdapterId` is ignored because the registry always provides a direct adapter.)

IceGrid.Registry.LMDB.MapSize

Synopsis

`IceGrid.Registry.LMDB.MapSize=`*`num`*

Description

Specifies the map size for the IceGrid LMDB database environment. The value is specified in megabytes. If not specified or set to 0, IceGrid uses a system-dependent default: 10 MB on Windows, and 100 MB on other platforms.

IceGrid.Registry.LMDB.Path

Synopsis

`IceGrid.Registry.LMDB.Path=`*`path`*

Description

Specifies the path of IceGrid registry LMDB database environment. The directory specified in *`path`* must exist - the IceGrid registry does not create this directory.

IceGrid.Registry.NodeSessionTimeout

Synopsis

`IceGrid.Registry.NodeSessionTimeout=`*`num`*

Description

Each IceGrid node establishes a session with the registry that must be refreshed periodically. If a node does not refresh its session within *`num`* seconds, the node's session is destroyed and the servers deployed on that node become unavailable to new clients. If not specified, the default value is 30 seconds.

IceGrid.Registry.PermissionsVerifier

Synopsis

`IceGrid.Registry.PermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::PermissionsVerifier` interface for controlling access to IceGrid sessions. The IceGrid registry invokes this proxy to validate each new client session created by a client with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using properties.

IceGrid.Registry.ReplicaName

Synopsis

`IceGrid.Registry.ReplicaName=`*`name`*

Description

Specifies the name of a registry replica. If not defined, the default value is `Master`, which is the name reserved for the master replica. Each registry replica must have a unique name.

IceGrid.Registry.ReplicaSessionTimeout

Synopsis

`IceGrid.Registry.ReplicaSessionTimeout=`*`num`*

Description

Each IceGrid registry replica establishes a session with the master registry that must be refreshed periodically. If a replica does not refresh

its session within *num* seconds, the replica's session is destroyed and the replica no longer receives replication information from the master registry. If not specified, the default value is 30 seconds.

IceGrid.Registry.Server.*AdapterProperty*

Synopsis

`IceGrid.Registry.Server.`*`AdapterProperty`*`=`*`value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.Server` for the object adapter that processes incoming requests from servers. Therefore, adapter properties can be used to configure this adapter. (Note any setting of `IceGrid.Registry.Server.AdapterId` is ignored because the registry always provides a direct adapter.)

IceGrid.Registry.SessionFilters

Synopsis

`IceGrid.Registry.SessionFilters=`*`num`*

Description

This property controls whether IceGrid establishes filters for sessions created with the IceGrid session manager. If *num* is set to a value larger than zero, IceGrid establishes these filters, so Glacier2 limits access to the `IceGrid::Query` and `IceGrid::Session` objects, and to objects and adapters allocated by the session. If *num* is set to zero, IceGrid does not establish filters, so access to objects is controlled solely by Glacier2's configuration.

The default value is 0.

IceGrid.Registry.SessionManager.*AdapterProperty*

Synopsis

`IceGrid.Registry.SessionManager.`*`AdapterProperty`*`=`*`value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.SessionManager` for the object adapter that processes incoming requests from client sessions. Therefore, adapter properties can be used to configure this adapter. (Note any setting of `IceGrid.Registry.SessionManager.AdapterId` is ignored because the registry always provides a direct adapter.)

For security reasons, defining endpoints for this object adapter is optional. If you do define endpoints, they should only be accessible to Glacier2 routers used to create IceGrid client sessions.

IceGrid.Registry.SessionTimeout

Synopsis

`IceGrid.Registry.SessionTimeout=`*`num`*

Description

IceGrid clients or administrative clients might establish a session with the registry. This session must be refreshed periodically. If the client does not refresh its session within *num* seconds, the session is destroyed. If not specified, the default value is 30 seconds.

IceGrid.Registry.SSLPermissionsVerifier

Synopsis

`IceGrid.Registry.SSLPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLPermissionsVerifier` interface for controlling access to IceGrid sessions. The IceGrid registry invokes this proxy to validate each new client session created by a client from a secure connection with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using properties.

IceGrid.Registry.Trace.Adapter

Synopsis

`IceGrid.Registry.Trace.Adapter=`*`num`*

Description

The object adapter trace level:

`0`	No object adapter trace (default).
`1`	Trace object adapter registration, removal, and replication.

IceGrid.Registry.Trace.Admin

Synopsis

`IceGrid.Registry.Trace.Admin=`*`num`*

Description

Set the trace level for the routing of operations to Ice.Admin objects through this registry.

0	No admin trace (default).
1	Trace routing of operations to Ice.Admin objects

IceGrid.Registry.Trace.Application

Synopsis

`IceGrid.Registry.Trace.Application=`*`num`*

Description

The application trace level:

0	No application trace (default).
1	Trace application addition, update, and removal.

IceGrid.Registry.Trace.Discovery

Synopsis

`IceGrid.Registry.Trace.Discovery=`*`num`*

Description

The discovery trace level:

`0`	No discovery trace (default).
`1`	Trace replied discovery lookup requests.
`2`	Like `1`, also includes discarded lookup requests.

IceGrid.Registry.Trace.Locator

Synopsis

`IceGrid.Registry.Trace.Locator=`*`num`*

Description

The locator and locator registry trace level:

0	No locator trace (default).
1	Trace failures to locate an adapter or object, and failures to register adapter endpoints.
2	Like 1, but more verbose, including registration of adapter endpoints.

IceGrid.Registry.Trace.Node

Synopsis

`IceGrid.Registry.Trace.Node=`*`num`*

Description

The node trace level:

0	No node trace (default).
1	Trace node registration, removal.
2	Like 1, but more verbose, including load statistics.

IceGrid.Registry.Trace.Object

Synopsis

`IceGrid.Registry.Trace.Object=`*`num`*

Description

The object trace level:

0	No object trace (default).
1	Trace object registration, removal.

IceGrid.Registry.Trace.Patch

Synopsis

`IceGrid.Registry.Trace.Patch=`*`num`*

Description

The patch trace level:

0	No patching trace (default).
1	Show summary of patch progress.

IceGrid.Registry.Trace.Replica

Synopsis

`IceGrid.Registry.Trace.Replica=`*`num`*

Description

The server trace level:

0	No server trace (default).
1	Trace session lifecycle between master replica and slaves.

IceGrid.Registry.Trace.Server

Synopsis

`IceGrid.Registry.Trace.Server=`*`num`*

Description

The server trace level:

0	No server trace (default).
1	Trace the addition and removal of servers in the Registry database.

IceGrid.Registry.Trace.Session

Synopsis

`IceGrid.Registry.Trace.Session=`*`num`*

Description

The session trace level:

0	No client or admin session trace (default).
1	Trace client or admin session registration, removal.
2	Like 1, but more verbose, includes keep alive messages.

IceGrid.Registry.UserAccounts

Synopsis

`IceGrid.Registry.UserAccounts=`*`file`*

Description

Specifies the file name of an IceGrid registry user account map file. Each line of the file must contain an identifier and a user account, separated by white space. The identifier will be matched against the client session identifier (the user ID for sessions created with a user ID and password, or the distinguished name for sessions created from a secure connection). This user account map file is used by IceGrid nodes to map session identifiers to user accounts if the nodes' `IceGrid.Node.UserAccountMapper` property is set to the proxy `IceGrid/RegistryUserAccountMapper`.

IceGridAdmin.*

On this page:

IceGridAdmin.AuthenticateUsingSSL

Synopsis

`IceGridAdmin.AuthenticateUsingSSL=`*`num`*

Description

If *`num`* is a value greater than zero, `icegridadmin` uses SSL authentication when establishing its session with the IceGrid registry. If not defined or the value is zero, `icegridadmin` uses user name and password authentication.

IceGridAdmin.Discovery.Address

Synopsis

`IceGridAdmin.Discovery.Address=`*`addr`*

Description

Specifies the multicast IP address to use for sending multicast discovery queries. If not defined, the default value depends on the setting of `Ice.IPv4`: if enabled (the default), the client uses the address `239.255.0.1`, otherwise the client assumes it should use IPv6 and defaults to the address `ff15::1` instead. This property is used to compose the value of `IceGridAdmin.Discovery.Lookup`.

IceGridAdmin.Discovery.Interface

Synopsis

`IceGridAdmin.Discovery.Interface=`*`intf`*

Description

Specifies the IP address of the interface to use for sending multicast discovery queries. If not defined, the discovery will use all the network interfaces available on the system to send UDP multicast datagrams. This property is used to compose the value of `IceGridAdmin.Discovery.Lookup` and `IceGridAdmin.Discovery.Reply.Endpoints`.

IceGridAdmin.Discovery.Lookup

Synopsis

`IceGridAdmin.Discovery.Lookup=`*`endpoints`*

Description

Specifies the endpoints that the client uses to send multicast discovery queries. If not defined, the endpoint is composed as follows:

```
udp -h addr -p port [--interface intf]
```

where *addr* is the value of `IceGridAdmin.Discovery.Address`, *port* is the value of `IceGridAdmin.Port`, and *intf* is the value of `IceGridAdmin.Discovery.Interface`. If multiple endpoints are defined, the queries will be sent on each endpoint.

IceGridAdmin.Discovery.Reply.*AdapterProperty*

Synopsis

`IceGridAdmin.Discovery.Reply.`*`AdapterProperty=value`*

Description

The client creates an object adapter named `IceGridAdmin.Discovery.Reply` for receiving replies to multicast discovery queries. If not otherwise defined by `IceGridAdmin.Discovery.Reply.Endpoints`, the endpoint for this object adapter is composed as follows:

```
udp [-h intf]
```

where *intf* is the value of `IceGridAdmin.Discovery.Interface`. A fixed port is not necessary for this endpoint.

You don't normally need to set other properties for this object adapter.

IceGridAdmin.Host

Synopsis

`IceGridAdmin.Host=`*`host`*

Description

When used together with `IceGridAdmin.Port`, `icegridadmin` connects directly to the target registry at the specified host and port.

IceGridAdmin.InstanceName

Synopsis

`IceGridAdmin.InstanceName=`*`name`*

Description

Specifies the name of an IceGrid instance to which `icegridadmin` will connect.

When using multicast discovery, you can define this property to limit your discovery results only to those locators deployed for the given instance, in case you have multiple unrelated IceGrid instances deployed that use the same multicast address and port.

IceGridAdmin.Password

Synopsis

`IceGridAdmin.Password=`*`password`*

Description

Specifies the password that `icegridadmin` should use when authenticating its session with the IceGrid registry. For security reasons you may prefer not to define a password in a plain-text configuration property, in which case you should omit this property and allow `icegridadmin` to prompt you for it interactively. This property is ignored when SSL authentication is enabled via `IceGridAdmin.AuthenticateUsi`

ngSSL.

IceGridAdmin.Port

Synopsis

`IceGridAdmin.Port=`*`port`*

Description

When used together with `IceGridAdmin.Host`, `icegridadmin` connects directly to the target registry at the specified host and port.

When using multicast discovery, this property specifies the port to use for sending multicast discovery queries. This property is also used to compose the value of `IceGridAdmin.Discovery.Lookup`.

If not set, the default value is `4061`.

IceGridAdmin.Replica

Synopsis

`IceGridAdmin.Replica=`*`name`*

Description

Specifies the name of the registry replica that `icegridadmin` should contact. If not defined, the default value is `Master`.

IceGridAdmin.Trace.Observers

Synopsis

`IceGridAdmin.Trace.Observers=`*`num`*

Description

If *`num`* is a value greater than zero, IceGrid GUI displays trace information about the observer callbacks it receives from the registry. If not defined, the default value is zero.

IceGridAdmin.Trace.SaveToRegistry

Synopsis

`IceGridAdmin.Trace.SaveToRegistry=`*`num`*

Description

If *`num`* is a value greater than zero, IceGrid GUI displays trace information about the modifications it commits to the registry. If not defined, the default value is zero.

IceGridAdmin.Username

Synopsis

`IceGridAdmin.Username=`*`name`*

Description

Specifies the username that `icegridadmin` should use when authenticating its session with the IceGrid registry. This property is ignored when SSL authentication is enabled via `IceGridAdmin.AuthenticateUsingSSL`.

IceLocatorDiscovery.*

This page describes the properties supported by the IceLocatorDiscovery plug-in.

On this page:

IceLocatorDiscovery.Address

Synopsis

`IceLocatorDiscovery.Address=`*`addr`*

Description

Specifies the multicast IP address to use for sending multicast discovery queries. If not defined, the default value depends on other property settings:

- If `Ice.PreferIPv6Address` is enabled or `Ice.IPv4` is disabled, IceLocatorDiscovery uses the IPv6 address `ff15::1`
- Otherwise IceLocatorDiscovery uses `239.255.0.1`

This property is used to compose the value of `IceLocatorDiscovery.Lookup`.

IceLocatorDiscovery.InstanceName

Synopsis

`IceLocatorDiscovery.InstanceName=`*`name`*

Description

Specifies the name of a locator instance. If you have multiple unrelated locators deployed that use the same multicast address and port, you can define this property to limit your discovery results only to those locators deployed for the given instance. If not defined, the plug-in adopts the instance name of the first locator to respond to a query; if a subsequent query discovers a locator with a different instance name, the plug-in logs a message and ignores the result.

IceLocatorDiscovery.Interface

Synopsis

`IceLocatorDiscovery.Interface=`*`intf`*

Description

Specifies the IP address of the interface to use for sending multicast discovery queries. If not defined, the discovery will use all the network interfaces available on the system to send UDP multicast datagrams. This property is used to compose the value of `IceLocatorDiscovery.Lookup` and `IceLocatorDiscovery.Reply.Endpoints`.

IceLocatorDiscovery.Locator.*AdapterProperty*

Synopsis

`IceLocatorDiscovery.Locator.`*`AdapterProperty=value`*

Description

IceLocatorDiscovery creates an object adapter named `IceLocatorDiscovery.Locator`, therefore all of the object adapter properties can be set.

You don't normally need to set properties for this object adapter.

IceLocatorDiscovery.Lookup

Synopsis

`IceLocatorDiscovery.Lookup=`*`endpoints`*

Description

Specifies the multicast endpoints that a client uses to send multicast discovery queries. If not defined, the endpoint is composed as follows:

```
udp -h addr -p port [--interface intf]
```

where *`addr`* is the value of `IceLocatorDiscovery.Address`, *`port`* is the value of `IceLocatorDiscovery.Port`, and *`intf`* is the value of `IceLocatorDiscovery.Interface`. If multiple endpoints are defined, the queries will be sent on each endpoint.

IceLocatorDiscovery.Port

Synopsis

`IceLocatorDiscovery.Port=`*`port`*

Description

Specifies the multicast port to use for sending multicast queries. If not set, the default value is `4061`.

IceLocatorDiscovery.Reply.*AdapterProperty*

Synopsis

`IceLocatorDiscovery.Reply.`*`AdapterProperty=value`*

Description

IceLocatorDiscovery creates an object adapter named `IceLocatorDiscovery.Reply` for receiving replies to multicast discovery queries. If not otherwise defined by `IceLocatorDiscovery.Reply.Endpoints`, the endpoint for this object adapter is composed as follows:

```
udp [-h intf]
```

where *`intf`* is the value of `IceLocatorDiscovery.Interface`. A fixed port is not necessary for this endpoint.

You don't normally need to set other properties for this object adapter.

IceLocatorDiscovery.RetryCount

Synopsis

`IceLocatorDiscovery.RetryCount=`*`num`*

Description

Specifies the maximum number of times that the plug-in will retry sending UDP multicast queries before giving up. The `IceLocatorDiscovery.Timeout` property determines how long the plug-in waits for a reply before trying again. If not defined, the default retry count is `3`.

IceLocatorDiscovery.RetryDelay

Synopsis

`IceLocatorDiscovery.RetryDelay=`*`num`*

Description

If the plug-in fails to receive any responses to a query after retrying the number of times specified by `IceLocatorDiscovery.RetryCount`, the plug-in waits at least *`num`* milliseconds before starting another round of query attempts. If not defined, the default value is `2000`.

IceLocatorDiscovery.Trace.Lookup

Synopsis

`IceLocatorDiscovery.Trace.Lookup=`*`num`*

Description

The lookup trace level:

`0`	No lookup trace (default).
`1`	Trace lookup success, failure or time out.
`2`	Like 1, but more verbose, also trace the lookup invocations.

IceLocatorDiscovery.Timeout

Synopsis

`IceLocatorDiscovery.Timeout=`*`num`*

Description

Specifies the time interval in milliseconds to wait for replies to UDP multicast queries. If no server replies during this time interval, the client will retry the request the number of times specified by `IceLocatorDiscovery.RetryCount`. If not defined, the default timeout is `300`.

IceMX.Metrics.*

On this page:

Metrics view are configured with the properties described below. The *view* below can be replaced with one of the following:

- IceMX.Metrics.*view-name*
- IceMX.Metrics.*view-name*.Map.*map-name*
- IceMX.Metrics.*view-name*.Map.*map-name*.Map.*submap-name*

If a view is defined without Map properties, the view will contain all the metrics map known by the Metrics facet. If a view defines one or more map properties it will only contain these maps.

For a list of supported maps see:

- The Metrics Facet
- Glacier2 Metrics
- IceStorm Metrics

IceMX.Metrics.*view*.Accept.*attribute*

Synopsis

`IceMX.Metrics.`*`view`*`.Accept.`*`attribute`*`=`*`regexp`*

Description

This property defines a rule to accept the monitoring of an instrumented object or operation based on the value of one of its attribute. If the *attribute* matches the specified *regex* and if it satisfies other `Accept` and `Reject` filters the instrumented object or operation will be monitored.

For example, to accept monitoring instrumented objects or operations which are from the object adapter named "MyAdapter", you can setup the following accept property:

- `IceMX.Metrics.MyView.Accept.parent=MyAdapter`

IceMX.Metrics.*view*.Disabled

Synopsis

`IceMX.Metrics.`*`view`*`.Disabled=`*`num`*

Description

If *num* is set to a value larger than zero, the metrics view or the map is disabled. This property is useful to pre-configure a view or map. The view can be disabled initially to not incur overhead and enabled only when needed at runtime.

IceMX.Metrics.*view*.GroupBy

Synopsis

`IceMX.Metrics.`*`view`*`.GroupBy=`*`delimited attributes`*

Description

This property defines how metrics are grouped and how the ID of each metrics object is created. The grouping is based on attributes specific

to the instrumented object or operation. For example, you can group the invocation metrics by operation name or proxy identity. All the invocations with the same operation name or proxy identity will record metrics using the same metrics object. You can specify several attributes to group metrics based on multiple attributes. You must delimit the attributes with delimiters when specify the value of the GroupBy property. A delimiter is any character which is not an alpha numeric or the dot character. Attributes which can be used to specify the value of this property are defined in relevant section of the Ice manual. Here are some examples of GroupBy properties.

- `IceMX.Metrics.MyView.GroupBy=operation`
- `IceMX.Metrics.MyView.GroupBy=identity [operation]`
- `IceMX.Metrics.MyView.GroupBy=remoteHost:remotePort`

IceMX.Metrics.*view*.Reject.*attribute*

Synopsis

`IceMX.Metrics.`*`view`*`.Reject.`*`attribute`*`=`*`regexp`*

Description

This property defines a rule to accept the monitoring of an instrumented object or operation based on the value of one of its attribute. If the *`attribute`* matches the specified *`regex`* and if it satisfies other `Accept` and `Reject` filters the instrumented object or operation will be monitored.

For example, to reject monitoring instrumented objects or operations which are from the object adapter named "Ice.Admin", you can setup the following reject property:

- `IceMX.Metrics.MyView.Reject.parent=Ice\.Admin`

IceMX.Metrics.*view*.RetainDetached

Synopsis

`IceMX.Metrics.`*`view`*`.RetainedDetached=`*`num`*

Description

If *`num`* is set to a value larger than zero, up to `num` metrics object whose `current` value is 0 will be kept in memory by the metrics map. This is useful to prevent indefinite memory growth if the monitoring of an instrumented object or operation creates a unique metrics object, only the last *`num`* metrics object will be kept in memory. The default value is `10`, meaning that at most ten metrics object with a `current` value equal to 0 will be retained by the metrics map.

IcePatch2.*

On this page:

IcePatch2.*AdapterProperty*

Synopsis

`IcePatch2.`*`AdapterProperty`*`=`*`value`*

Description

IcePatch2 uses the adapter name `IcePatch2` for the server. Therefore, adapter properties can be used to configure this adapter.

IcePatch2.Directory

Synopsis

`IcePatch2.Directory=`*`dir`*

Description

The IcePatch2 server uses this property to determine the data directory if no data directory is specified on the command line.

This property is also used by IcePatch2 clients to determine the local data directory.

IcePatch2.InstanceName

Synopsis

`IcePatch2.InstanceName=`*`name`*

Description

Specifies the identity category for well-known IcePatch2 objects. If defined, the identity of the `IcePatch2::Admin` interface becomes *`name`*`/admin` and the identity of the `IcePatch2::FileServer` interface becomes *`name`*`/server`.

If not defined, the default value is `IcePatch2`.

IcePatch2Client.*

On this page:

IcePatch2Client.ChunkSize

Synopsis

`IcePatch2Client.ChunkSize=`*`kilobytes`*

Description

The IcePatch2 client uses this property to determine how many kilobytes are retrieved with each call to `getFileCompressed`.

The default value is `100`.

IcePatch2Client.Directory

Synopsis

`IcePatch2Client.Directory=`*`dir`*

Description

The IcePatch2 client uses this property to determine the local data directory.

IcePatch2Client.Proxy

Synopsis

`IcePatch2Client.Proxy=`*`proxy`*

Description

The IcePatch2 client uses this property to locate the IcePatch2 server.

IcePatch2Client.Remove

Synopsis

`IcePatch2Client.Remove=`*`num`*

Description

This property determines whether IcePatch2 clients delete files that exist locally, but not on the server. A negative or zero value prevents removal of files. A value of `1` enables removal and causes the client to halt with an error if removal of a file fails. A value of `2` or greater also enables removal, but causes the client to silently ignore errors during removal.

The default value is `1`.

IcePatch2Client.Thorough

Synopsis

`IcePatch2Client.Thorough=`*`num`*

Description

This property determines whether IcePatch2 clients recompute checksums. Any value greater than zero is interpreted as true. The default value is `0` (false).

IceSSL.*

On this page:

IceSSL Property Overview

The IceSSL implementations for our supported platforms use many of the same configuration properties. However, there are some properties that are specific to certain platforms or languages. For properties with such limitations, we list the supported platforms or underlying SSL libraries in the synopsis and provide additional platform-specific notes if necessary. You'll see the following platforms, languages and SSL libraries listed in the property reference:

- SChannel (C++ on Windows)
- SecureTransport (C++ and Objective-C on macOS and iOS)
- OpenSSL (C++ on Linux or Windows)
- Java
- .NET
- UWP

A property is supported by all platforms if no limitations are mentioned.

Finally, note that Ice for Objective-C and the Ice extensions for Python, Ruby and PHP use IceSSL for C++, therefore they use the IceSSL properties for SChannel, SecureTransport or OpenSSL as appropriate for the target platform.

> **JavaScript**
> These IceSSL properties have no effect on JavaScript.

IceSSL.Alias

Synopsis

`IceSSL.Alias=`*`alias`* (Java)

Description

Selects a particular certificate from the key store specified by `IceSSL.Keystore`. The certificate identified by *`alias`* is presented to the peer request during authentication.

IceSSL.CAs

Synopsis

`IceSSL.CAs=`*`path`* (SChannel, SecureTransport, OpenSSL, .NET)

Description

Specifies the path name of a file containing the certificates of trusted certificate authorities (CAs).

If you wish to use the CA certificates bundled with your platform, leave this property unset and enable `IceSSL.UsePlatformCAs`.

Platform Notes

SChannel, SecureTransport, .NET

The file can be encoded using the DER or PEM formats. When using PEM, the file can contain multiple certificates.

IceSSL attempts to locate *`path`* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *`path`* relative to the default directory defined by `IceSSL.DefaultDir`.

On iOS, IceSSL also attempts to open the specified CA certificate file as `Resources/`*`DefaultDir/path`* in the application's resource bundle if IceSSL.DefaultDir is defined or as `Resources/`*`path`* if not defined.

OpenSSL

The file must be encoded using the PEM format and can contain multiple certificates. The *`path`* can also refer to a directory prepared in advance using the OpenSSL utility `c_rehash`.

IceSSL attempts to locate *`path`* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *`path`* relative to the default directory defined by `IceSSL.DefaultDir`.

Java

See `IceSSL.Truststore`.

UWP

This property has no effect in UWP applications. UWP application must set the "Exclusive Trust" flag in the "Certificates" declaration of the application manifest and add the required certificates to the applications "Root" store, or unset the "Exclusive Trust" flag and rely on the system "Trusted Certificate Authorities" current user store.

IceSSL.CertFile

Synopsis

`IceSSL.CertFile=`*`file`* (SecureTransport, .NET, UWP)

`IceSSL.CertFile=`*`file`*`[;`*`file`*`]` (SChannel)
`IceSSL.CertFile=`*`file`*`[:`*`file`*`]` (OpenSSL)

Description

Specifies a file that contains the program's certificate and the corresponding private key. The file name may be specified relative to the default directory defined by `IceSSL.DefaultDir`.

Platform Notes

SChannel

The file must use the PFX (PKCS#12) format and contain the certificate and its private key. If a password is required to load the file, the application must either install a password handler or supply the password using `IceSSL.Password`, otherwise IceSSL will reject the certificate.

This property accepts certificates for both RSA and DSA. To specify both certificates, separate the file names using the platform's path separator character.

IceSSL attempts to locate *`file`* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *`file`* relative to the default directory defined by `IceSSL.DefaultDir`.

SecureTransport

The file must use the PFX (PKCS#12) format and contain the certificate and its private key. If a password is required to load the file, macOS will use its default graphical password prompt unless the application has installed a password handler or supplied the password using `IceSSL.Password`. Define `IceSSL.Keychain` to import this certificate into the specified keychain.

IceSSL attempts to locate *`file`* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *`file`* relative to the default directory defined by `IceSSL.DefaultDir`.

On iOS, IceSSL also attempts to open the specified certificate file as `Resources/`*`DefaultDir/file`* in the application's resource bundle if IceSSL.DefaultDir is defined or as `Resources/`*`file`* if not defined.

OpenSSL

The file must use the PFX (PKCS#12) format and contain the certificate and its private key. If a password is required to load the file, OpenSSL will prompt the user at the terminal unless the application has installed a password handler or supplied the password using `IceSSL.Password`.

This property accepts certificates for both RSA and DSA. To specify both certificates, separate the file names using the platform's path separator character.

IceSSL attempts to locate *`file`* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *`file`* relative to the default directory defined by `IceSSL.DefaultDir`.

.NET

The file must use the PFX (PKCS#12) format and contain the certificate and its private key. The password for the file must be supplied using `IceSSL.Password`.

IceSSL attempts to locate *`file`* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *`file`* relative to the default directory defined by `IceSSL.DefaultDir`.

Java

See `IceSSL.Keystore`.

UWP

The file must use the PFX (PKCS#12) format and contain the certificate and its private key. The password for the file must be supplied using `IceSSL.Password`. The certificate is imported into the application `Personal` certificate store. The file must use one of `ms-appx:///` or `ms-appdata:///` URIs schemes.

IceSSL.CertStore

Synopsis

`IceSSL.CertStore=`*`name`* (SChannel, .NET, UWP)

Description

Specifies the name of a certificate store to use when locating certificates via `IceSSL.FindCert`. Legal values for *`name`* include `AddressBook`, `AuthRoot`, `CertificateAuthority`, `Disallowed`, `My`, `Root`, `TrustedPeople`, and `TrustedPublisher`. You can also use an arbitrary value for *`name`*.

If not specified, the default value is `My`.

IceSSL.CertStoreLocation

Synopsis

`IceSSL.CertStoreLocation=CurrentUser|LocalMachine` (SChannel, .NET)

Description

Specifies the location of a certificate store to use when locating certificates via `IceSSL.FindCert`. If not specified, the default value is `CurrentUser`.

> An Ice program running as a Windows service will typically need to set this property to `LocalMachine`.

IceSSL.CertVerifier

Synopsis

`IceSSL.CertVerifier=`*`classname`* (Java, .NET)

Description

Specifies the name of a Java or .NET class that implements the `IceSSL.CertificateVerifier` interface for performing application-defined certificate verification.

Platform Notes

SChannel, SecureTransport, OpenSSL and UWP

C++ applications can install a certificate verifier programmatically.

IceSSL.CheckCertName

Synopsis

`IceSSL.CheckCertName=`*`num`*

Description

If *`num`* is a value greater than zero, IceSSL will enable the host name verification mechanism of the underlying SSL engine. IceSSL skips this validation step if the server does not supply a certificate, or if the proxy endpoint does not include a host name and `Ice.Default.Host` is not defined. This property has no effect on a server's validation of a client's certificate. If no match is found, IceSSL aborts the connection attempt and raises an exception. If not defined, the default value is zero.

IceSSL.CheckCRL

Synopsis

`IceSSL.CheckCRL=`*`num`* (.NET)

Description

If *num* is a value greater than zero, IceSSL checks the certificate revocation list (CRL) to determine if the peer's certificate has been revoked. The value for *num* determines the resulting behavior:

`0`	Disables CRL checking.
`1`	If a certificate is revoked, IceSSL aborts the connection, logs a message and raises an exception. If a certificate's revocation status is unknown, IceSSL logs a message but accepts the certificate.
`2`	If a certificate is revoked or its revocation status is unknown, IceSSL aborts the connection, logs a message and raises an exception.

The `IceSSL.Trace.Security` property must be set to a non-zero value to see CRL-related log messages. If `IceSSL.CheckCRL` is not defined, the default value is zero.

IceSSL.Ciphers

Synopsis

`IceSSL.Ciphers=`*`ciphers`* (SChannel, SecureTransport, OpenSSL, Java)

Description

Specifies the cipher suites that IceSSL is allowed to negotiate. A cipher suite is a set of algorithms that satisfies the four requirements for establishing a secure connection: signing and authentication, key exchange, secure hashing, and encryption. Some algorithms satisfy more than one requirement, and there are many possible combinations.

If not specified, the plug-in uses the security provider's default cipher suites. Enable `IceSSL.Trace.Security` and carefully review the application's log output to determine which cipher suites are enabled by default, or to verify your cipher suite configuration.

Platform Notes

SChannel

The value of this property is a whitespace-separated list that can include any of the following:

- `3DES`
- `AES_128`
- `AES_256`
- `DES`
- `RC2`
- `RC4`

For example, the following setting enables AES cipher suites:

```
IceSSL.Ciphers=AES_128 AES_256
```

Anonymous Diffie Hellman ciphers are not supported on Windows.

SecureTransport

The property value is interpreted as a list of tokens delimited by white space. The plug-in executes the tokens in the order of appearance in order to assemble the list of enabled cipher suites. The table below describes the tokens:

`ALL`	Reserved keyword that enables all supported cipher suites. If specified, it must be the first token in the list. Use with caution as it may enable low-security cipher suites.
`NONE`	Reserved keyword that disables all cipher suites. If specified, it must be the first token in the list. Use `NONE` to start with an empty set of cipher suites and then add only those suites you want to allow.
NAME	Includes the cipher suite whose name matches *NAME* exactly.
`!`*NAME*	Excludes the cipher suite whose name matches *NAME* exactly.
(*EXP*)	Includes all cipher suites whose names contain the given regular expression *EXP*.
`!`(*EXP*)	Excludes all cipher suites whose names contain the given regular expression *EXP*.

For example, the following setting disables all cipher suites except those that support 256-bit AES with SHA256:

```
IceSSL.Ciphers=NONE (AES_256.*SHA256)
```

Note that no warning is given if an unrecognized cipher is specified.

OpenSSL

The value of this property is passed directly to the OpenSSL library and the list of supported ciphers depends on how your installation of OpenSSL was compiled. You can obtain a complete list of the supported cipher suites using the command `openssl ciphers`. This command will likely generate a long list. To simplify the selection process, OpenSSL supports several classes of ciphers. Classes and ciphers can be excluded by prefixing them with an exclamation point. The special keyword `@STRENGTH` sorts the current cipher list in order of encryption algorithm key length. The classes are:

`ALL`	Enables all supported cipher suites. This class should be used with caution, as it may enable low-security cipher suites.
`ADH`	Anonymous ciphers.
`LOW`	Low bit-strength ciphers.
`EXP`	Export-crippled ciphers.

Here is an example of a reasonable setting:

```
IceSSL.Ciphers=ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH
```

This value excludes the ciphers with low bit-strength and known problems, and orders the remaining ciphers according to their key length. Note that no warning is given if an unrecognized cipher is specified.

Java

The property value is interpreted as a list of tokens delimited by white space. The plug-in executes the tokens in the order of appearance in order to assemble the list of enabled cipher suites. The table below describes the tokens:

`ALL`	Reserved keyword that enables all supported cipher suites. If specified, it must be the first token in the list. Use with caution as it may enable low-security cipher suites.
`NONE`	Reserved keyword that disables all cipher suites. If specified, it must be the first token in the list. Use `NONE` to start with an empty set of cipher suites and then add only those suites you want to allow.
NAME	Enables the cipher suite matching the given name.
`!`*NAME*	Disables the cipher suite matching the given name.

(EXP)	Enables cipher suites whose names contain the regular expression *EXP*. For example, the value `NONE (.*DH_anon.*AES.*)` selects only cipher suites that use anonymous Diffie-Hellman authentication with AES encryption.
`!(`*EXP*`)`	Disables cipher suites whose names contain the regular expression *EXP*. For example, the value `ALL !(.*DH_anon.*AES.*)` enables all cipher suites except those that use anonymous Diffie-Hellman authentication with AES encryption.

IceSSL.DefaultDir

Synopsis

`IceSSL.DefaultDir=`*`path`*

Description

Specifies the default directory in which to look for certificates, key stores, and other files. See the descriptions of the relevant properties for more information.

Platform Notes

UWP

This property has not effect in UWP application.

IceSSL.DH.*bits*

Synopsis

`IceSSL.DH.`*`bits`*`=`*`file`* (OpenSSL)

Description

Specifies a *`file`* containing Diffie Hellman parameters whose key length is *`bits`*, as shown in the following example:

```
IceSSL.DH.1024=dhparams1024.pem
```

IceSSL supplies default parameters for key lengths of 512, 1024, 2048, and 4096 bits, which are used if no user-defined parameters of the desired key length are specified. The parameters must be encoded using the PEM format.

IceSSL attempts to locate *`file`* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *`file`* relative to the default directory defined by `IceSSL.DefaultDir`.

Platform Notes

SecureTransport

See `IceSSL.DHParams`.

IceSSL.DHParams

Synopsis

`IceSSL.DHParams=`*`file`* (SecureTransport)

Description

Specifies a *file* containing Diffie Hellman parameters. The parameters must be encoded using the DER format.

This property only affects server (incoming) connections. Clients obtain their DH parameters when negotiating a SSL/TLS connection with the server.

IceSSL attempts to locate *file* as specified; if the given path is relative but does not exist, IceSSL also attempts to locate *file* relative to the default directory defined by `IceSSL.DefaultDir`.

> If this property is not specified, macOS will generate its own set of Diffie Hellman parameters for the process. Computing these parameters at run time can take up to 30 seconds, so we recommend generating them in advance and defining this property.

Diffie Hellman parameters are not supported on iOS, this property will be ignored.

> You can generate Diffie Hellman parameters using the `openssl dhparam` command.

Platform Notes

OpenSSL

Use `IceSSL.DH.bits`.

IceSSL.EntropyDaemon

Synopsis

`IceSSL.EntropyDaemon=file` (OpenSSL)

Description

Specifies a Unix domain socket for the entropy gathering daemon, from which OpenSSL gathers entropy data to initialize its random number generator.

IceSSL.FindCert

Synopsis

`IceSSL.FindCert=criteria` (SChannel, SecureTransport, .NET)

Description

Builds a collection of certificates that will be used for authentication.

A server requires a certificate for authentication purposes, therefore IceSSL selects the first certificate in the accumulated collection. This is normally the certificate loaded via `IceSSL.CertFile`, if that property was defined. Otherwise, IceSSL selects one of the certificates identified by `IceSSL.FindCert`.

Platform Notes

SChannel, .NET

IceSSL queries a certificate store for matching certificates and adds them to the application's certificate collection. The settings for `IceSSL.CertStore` and `IceSSL.CertStoreLocation` determine the target certificate store to be queried.

The value for *criteria* may be `*`, in which case all of the certificates in the store are selected. Otherwise, *criteria* must be one or more *field*`:`*value* pairs separated by white space. The valid field names are described below:

`Issuer`	Matches a substring of the issuer's name.

`IssuerDN`	Matches the issuer's entire distinguished name.
`Serial`	Matches the certificate's serial number.
`Subject`	Matches a substring of the subject's name.
`SubjectDN`	Matches the subject's entire distinguished name.
`SubjectKeyId`	Matches the certificate's subject key identifier.
`Thumbprint`	Matches the certificate's SHA1 hash.

The field names are case-insensitive. If multiple criteria are specified, only certificates that match all criteria are selected. Values must be enclosed in single or double quotes to preserve white space.

SecureTransport

IceSSL queries the keychain for matching certificates and adds them to the application's certificate collection. IceSSL uses the keychain identified in `IceSSL.Keychain`, or the user's default keychain if `IceSSL.Keychain` is not defined.

The value for *`criteria`* must be one or more *`field`*`:`*`value`* pairs separated by white space. The valid field names are described below:

`Label`	Matches the user-visible label.
`Serial`	Matches the certificate's serial number.
`Subject`	Matches a substring of the subject's name.
`SubjectKeyId`	Matches the certificate's subject key identifier.

The field names are case-insensitive. If multiple criteria are specified, only certificates that match all criteria are selected. Values must be enclosed in single or double quotes to preserve white space.

On iOS, matching on the `Subject` field is not supported.

Java

Use `IceSSL.Alias`.

IceSSL.InitOpenSSL

Synopsis

`IceSSL.InitOpenSSL=`*`num`* (OpenSSL)

Description

Indicates whether IceSSL should perform the global initialization tasks for the OpenSSL library. The default value is 1, meaning IceSSL will initialize OpenSSL. An application can set this property to zero if it wishes to perform the OpenSSL initialization itself, which can be useful when the application uses multiple components that depend on OpenSSL.

IceSSL.Keychain

Synopsis

`IceSSL.Keychain=`*`name`* (SecureTransport)

Description

Specifies the name of a keychain in which to import the certificate identified by `IceSSL.CertFile`. Set `IceSSL.KeychainPassword` if the specified keychain has a password.

A relative path name is opened relative to the current working directory. If the specified keychain file does not exist, a new file is created. If not defined, IceSSL uses the user's default keychain.

On iOS this property is ignored, IceSSL uses the default device keychain.

IceSSL.KeychainPassword

Synopsis

`IceSSL.KeychainPassword=`*`password`* (SecureTransport)

Description

Specifies the password for the keychain identified by `IceSSL.Keychain`. If not defined, IceSSL attempts to open the keychain without a password.

On iOS, this property is ignored.

IceSSL.Keystore

Synopsis

`IceSSL.Keystore=`*`file`* (Java)

Description

Specifies a key store file containing certificates and their private keys. If the key store contains multiple certificates, you should specify a particular one to use for authentication using `IceSSL.Alias`. IceSSL first attempts to open *`file`* as a class loader resource and then as a regular file. If the given path is relative but does not exist, IceSSL also attempts to locate it relative to the default directory defined by `IceSSL.DefaultDir`. The format of the file is determined by `IceSSL.KeystoreType`.

If this property is not defined, the application will not be able to supply a certificate during SSL handshaking. As a result, the application may not be able to negotiate a secure connection, or might be required to use an anonymous cipher suite.

IceSSL.KeystorePassword

Synopsis

`IceSSL.KeystorePassword=`*`password`* (Java)

Description

Specifies the password used to verify the integrity of the key store defined by `IceSSL.Keystore`. The integrity check is skipped if this property is not defined.

It is a security risk to use a plain-text password in a configuration file.

IceSSL.KeystoreType

Synopsis

`IceSSL.KeystoreType=`*`type`* (Java)

Description

Specifies the format of the key store file defined by `IceSSL.Keystore`. Legal values are `JKS` and `PKCS12`. If not defined, the JVM's default value is used (normally `JKS`).

IceSSL.Password

Synopsis

`IceSSL.Password=`*`password`*

Description

Specifies the password necessary to decrypt the private key.

> It is a security risk to use a plain-text password in a configuration file.

Platform Notes

SChannel, SecureTransport, OpenSSL, UWP

This property supplies the password that was used to secure the private key contained in the file defined by `IceSSL.CertFile`.

Java

This property supplies the password that was used to secure the private key contained in the key store defined by `IceSSL.Keystore`. All of the keys in the key store must use the same password.

.NET

This property supplies the password that was used to secure the file defined by `IceSSL.CertFile`.

iOS

This property supplies the password that was used to secure the file defined by `IceSSL.CertFile`.

IceSSL.PasswordCallback

Synopsis

`IceSSL.PasswordCallback=`*`classname`* (Java, .NET)

Description

Specifies the name of a Java or .NET class that implements the `IceSSL.PasswordCallback` interface. Using a password callback is a more secure alternative to specifying a password in a plain-text configuration file.

Platform Notes

SChannel, SecureTransport, OpenSSL, UWP

Use `setPasswordPrompt` to install a password callback in the plug-in.

IceSSL.PasswordRetryMax

Synopsis

`IceSSL.PasswordRetryMax=`*`num`* (SChannel, SecureTransport, OpenSSL, UWP)

Description

Specifies the number of attempts IceSSL should allow the user to make when entering a password. If not defined, the default value is 3.

IceSSL.Protocols

Synopsis

`IceSSL.Protocols=`*`list`* (SChannel, OpenSSL, Java, .NET)

Description

Specifies the protocols to allow during SSL handshaking. Legal values are as follows:

- `SSL3`, `SSLv3`
- `TLS1`, `TLSv1`
- `TLS1_0`, `TLSv1_0` (aliases for `TLS1`)
- `TLS1_1`, `TLSv1_1`
- `TLS1_2`, `TLSv1_2`

Case is ignored, so for example `tls1_2` is also accepted.

You can specify multiple values for this property by separating them with commas or white space. If this property is not defined, the default setting is as follows:

```
IceSSL.Protocols=TLS1_0, TLS1_1, TLS1_2
```

Platform Notes

SecureTransport

Use `IceSSL.ProtocolVersionMin` and `IceSSL.ProtocolVersionMax`.

IceSSL.ProtocolVersionMax

Synopsis

`IceSSL.ProtocolVersionMax=`*`prot`* (SecureTransport)

Description

Specifies the maximum protocol to allow during SSL handshaking. Legal values are the same as for `IceSSL.Protocols`. If this property is not defined, the system's default value is used.

Platform Notes

SChannel, OpenSSL, Java, .NET

Use `IceSSL.Protocols`.

IceSSL.ProtocolVersionMin

Synopsis

`IceSSL.ProtocolVersionMin=`*`prot`* (SecureTransport)

Description

Specifies the minimum protocol to allow during SSL handshaking. Legal values are the same as for `IceSSL.Protocols`. If this property is not defined, the default value is `TLS1_0`.

Platform Notes

SChannel, OpenSSL, Java, .NET

Use `IceSSL.Protocols`.

IceSSL.Random

Synopsis

`IceSSL.Random=`*`filelist`* (OpenSSL, Java)

Description

Specifies one or more files containing data to use when seeding the random number generator. The file names should be separated using the platform's path separator.

Platform Notes

OpenSSL

IceSSL attempts to locate each file as specified; if a given path is relative but does not exist, IceSSL also attempts to locate it relative to the default directory defined by `IceSSL.DefaultDir`.

Java

IceSSL first attempts to open each file as a class loader resource and then as a regular file. If a given path is relative but does not exist, IceSSL also attempts to locate it relative to the default directory defined by `IceSSL.DefaultDir`.

IceSSL.SchannelStrongCrypto

Synopsis

`IceSSL.SchannelStrongCrypto=`*`num`* (SChannel)

Description

If *`num`* is a value greater than zero, the IceSSL SChannel implementation will set the `SCH_USE_STRONG_CRYPTO` flag which instructs SChannel to disable known weak cryptographic algorithms. The default value for this property is 0 for better interoperability.

IceSSL.Trace.Security

Synopsis

`IceSSL.Trace.Security=`*`num`* (SChannel, SecureTransport, OpenSSL, Java, .NET)

Description

The SSL plug-in trace level:

`0`	No security tracing (default).
`1`	Display diagnostic information about SSL connections.

IceSSL.TrustOnly

Synopsis

`IceSSL.TrustOnly=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (SChannel, SecureTransport, OpenSSL, Java, .NET, UWP)

Description

Identifies trusted and untrusted peers. This family of properties provides an additional level of authentication by using the peer certificate's distinguished name (DN) to decide whether to accept or reject a connection.

Each *ENTRY* in the property value consists of relative distinguished name (RDN) components, formatted according to the rules in RFC 2253. Specifically, the components must be separated by commas, and any component that contains a comma must be escaped or enclosed in quotes. For example, the following two property definitions are equivalent:

```
IceSSL.TrustOnly=O="Acme, Inc.",OU=Sales
IceSSL.TrustOnly=O=Acme\, Inc.,OU="Sales"
```

Use a semicolon to separate multiple entries in a property:

```
IceSSL.TrustOnly=O=Acme\, Inc.,OU=Sales;O=Acme\, Inc.,OU=Marketing
```

By default, each entry represents an acceptance entry. A `!` character appearing at the beginning of an entry signifies a rejection entry. The order of the entries in a property is not important.

After the SSL engine has successfully completed its authentication process, IceSSL evaluates the relevant `IceSSL.TrustOnly` properties in an attempt to find an entry that matches the peer certificate's DN. For a match to be successful, the peer DN must contain an exact match for all of the RDN components in an entry. An entry may contain as many RDN components as you wish, depending on how narrowly you need to restrict access. The order of the RDN components in an entry is not important.

The connection semantics are described below:

1. IceSSL aborts the connection if any rejection or acceptance entries are defined and the peer does not supply a certificate.
2. IceSSL aborts the connection if the peer DN matches any rejection entry. (This is true even if the peer DN also matches an acceptance entry.)
3. IceSSL accepts the connection if the peer DN matches any acceptance entry, or if no acceptance entries are defined.

Our original example limits access to people in the sales and marketing departments:

```
IceSSL.TrustOnly=O=Acme\, Inc.,OU=Sales;O=Acme\, Inc.,OU=Marketing
```

If it later becomes necessary to deny access to certain individuals in these departments, you can add a rejection entry and restart the program:

```
IceSSL.TrustOnly=O=Acme\, Inc.,OU=Sales; O=Acme\, Inc.,OU=Marketing;
!O=Acme\, Inc.,CN=John Smith
```

While testing your trust configuration, you may find it helpful to set the `IceSSL.Trace.Security` property to a non-zero value, which causes IceSSL to display the DN of each peer during connection establishment.

This property affects incoming and outgoing connections. IceSSL also supports similar properties that affect only incoming connections or only outgoing connections.

Platform Notes

UWP

UWP applications can only match the CN part of the DN, the full DN is not provided by the UWP certificate APIs.

IceSSL.TrustOnly.Client

Synopsis

`IceSSL.TrustOnly.Client=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (SChannel, SecureTransport, OpenSSL, Java, .NET, UWP)

Description

Identifies trusted and untrusted peers for outgoing (client) connections. The entries defined in this property are combined with those of `IceSSL.TrustOnly`.

Platform Notes

UWP

UWP applications can only match the CN part of the DN, the full DN is not provided by the UWP certificate APIs.

IceSSL.TrustOnly.Server

Synopsis

`IceSSL.TrustOnly.Server=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (SChannel, SecureTransport, OpenSSL, Java, .NET)

Description

Identifies trusted and untrusted peers for incoming ("server") connections. The entries defined in this property are combined with those of `IceSSL.TrustOnly`. To configure trusted and untrusted peers for a particular object adapter, use `IceSSL.TrustOnly.Server.AdapterName`.

IceSSL.TrustOnly.Server.*AdapterName*

Synopsis

`IceSSL.TrustOnly.Server.`*`AdapterName`*`=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (SChannel, SecureTransport, OpenSSL, Java, .NET)

Description

Identifies trusted and untrusted peers for incoming (server) connections to the object adapter *`AdapterName`*. The entries defined in this property are combined with those of `IceSSL.TrustOnly` and `IceSSL.TrustOnly.Server`.

IceSSL.Truststore

Synopsis

`IceSSL.Truststore=`*`file`* (Java)

Description

Specifies a key store file containing the certificates of trusted certificate authorities. IceSSL first attempts to open *`file`* as a class loader resource and then as a regular file. If the given path is relative but does not exist, IceSSL also attempts to locate it relative to the default directory defined by `IceSSL.DefaultDir`. The format of the file is determined by `IceSSL.TruststoreType`.

If this property is not defined, IceSSL uses the value of `IceSSL.Keystore` by default. If no truststore is specified and the keystore does not contain a valid certificate chain, the application will not be able to authenticate the peer's certificate during SSL handshaking. As a result, the application may not be able to negotiate a secure connection, or might be required to use an anonymous cipher suite.

Platform Notes

SChannel, SecureTransport, OpenSSL, .NET

Use `IceSSL.CAs`.

IceSSL.TruststorePassword

Synopsis

`IceSSL.TruststorePassword=`*`password`* (Java)

Description

Specifies the password used to verify the integrity of the key store defined by `IceSSL.Truststore`. The integrity check is skipped if this property is not defined.

> It is a security risk to use a plain-text password in a configuration file.

Platform Notes

SChannel, SecureTransport, OpenSSL, .NET

Use `IceSSL.Password`.

IceSSL.TruststoreType

Synopsis

`IceSSL.TruststoreType=`*`type`* (Java)

Description

Specifies the format of the key store file defined by `IceSSL.Truststore`. Legal values are `JKS` and `PKCS12`. If not defined, the default value is `JKS`.

IceSSL.UsePlatformCAs

Synopsis

`IceSSL.UsePlatformCAs=`*`num`*

Description

If *`num`* is a value greater than zero, IceSSL uses the platform's bundled Root Certificate Authorities. This setting is ignored if `IceSSL.CAs` is defined.

If not defined, the default value is zero.

Platform Notes

UWP

This property has no effect in UWP applications. A UWP application must set the `Exclusive Trust` flag in the `Certificates` declaration of the application manifest to decide whether or not to trust certificates from the `Trusted Root Certificate Authorities` user store.

IceSSL.VerifyDepthMax

Synopsis

`IceSSL.VerifyDepthMax=`*`num`* (SChannel, SecureTransport, OpenSSL, Java, .NET, UWP)

Description

Specifies the maximum depth of a trusted peer's certificate chain, including the peer's certificate. A value of zero accepts chains of any length. If not defined, the default value is 3.

IceSSL.VerifyPeer

Synopsis

`IceSSL.VerifyPeer=`*`num`* (SChannel, SecureTransport, OpenSSL, Java, .NET, UWP)

Description

Specifies the verification requirements to use during SSL handshaking. The legal values are shown in the table below. If this property is not defined, the default value is `2`.

`0`	For an outgoing connection, the client verifies the server's certificate (if an anonymous cipher is not used) but does not abort the connection if verification fails. For an incoming connection, the server does not request a certificate from the client.
`1`	For an outgoing connection, the client verifies the server's certificate and aborts the connection if verification fails. For an incoming connection, the server requests a certificate from the client and verifies it if one is provided, aborting the connection if verification fails.
`2`	For an outgoing connection, the semantics are the same as for the value `1`. For an incoming connection, the server requires a certificate from the client and aborts the connection if verification fails.

Platform Notes

.NET

This property has no effect on outgoing connections, since .NET always uses the semantics of value `2`. For an incoming connection, the value `0` has the same semantics as the value `1`.

IceStorm Properties

IceStorm is an IceBox service that you can install using any name you like. For example:

```
IceBox.Service.DataFeed=IceStormService,...
```

The service name you choose is also used as the prefix for IceStorm's configuration properties. In the example above, the IceStorm configuration properties would use the `DataFeed` prefix, as in `DataFeed.Discard.Interval=10`.

In the property descriptions below, replace *`service`* with the service name from your IceStorm configuration.

On this page:

service.Discard.Interval

Synopsis

`service``.Discard.Interval=`*`num`*

Description

An IceStorm server detects when a subscriber to which it forwards events becomes non-functional and, at that point, stops delivery attempts to that subscriber for *`num`* seconds before trying to forward events to that subscriber again. The default value of this property is 60 seconds.

service.Election.ElectionTimeout

Synopsis

`service``.Election.ElectionTimeout=`*`num`*

Description

This property is used by a replicated IceStorm deployment. It specifies the interval in seconds at which a coordinator attempts to form larger

groups of replicas. If not defined, the default value is 10.

service.Election.MasterTimeout

Synopsis

`service.Election.MasterTimeout=num`

Description

This property is used by a replicated IceStorm deployment. It specifies the interval in seconds at which a slave checks the status of the coordinator. If not defined, the default value is 10.

service.Election.ResponseTimeout

Synopsis

`service.Election.ResponseTimeout=num`

Description

This property is used by a replicated IceStorm deployment. It specifies the interval in seconds that a replica waits for replies to an invitation to form a larger group. Lower priority replicas wait for intervals inversely proportional to the maximum priority:

```
ResponseTimeout + ResponseTimeout * (max - pri)
```

If not defined, the default value is 10.

service.Flush.Timeout

Synopsis

`service.Flush.Timeout=num`

Description

Defines the interval in milliseconds with which events are sent to batch subscribers. The default is 1000ms.

service.InstanceName

Synopsis

`service.InstanceName=name`

Description

Specifies an alternate identity category for all objects hosted by the IceStorm object adapters. If not specified, the default identity category is `IceStorm`.

service.LMDB.MapSize

Synopsis

```
service.LMDB.MapSize=num
```

Description

Specifies the map size for the IceStorm LMDB database environment. The value is specified in megabytes. If not specified or set to 0, IceStorm uses a system-dependent default: 10 MB on Windows, and 100 MB on other platforms.

service.LMDB.Path

Synopsis

```
service.LMDB.Path=dir
```

Description

Specifies the path to the LMDB database environment of this IceStorm service. If not specified, the default value is the service name. This directory must exist when IceStorm starts up unless IceStorm is in transient mode.

service.Node.*AdapterProperty*

Synopsis

```
service.Node.AdapterProperty=value
```

Description

In a replicated deployment, IceStorm uses the adapter name `service.Node` for the replica node's object adapter. Therefore, adapter properties can be used to configure this adapter.

service.NodeId

Synopsis

```
service.NodeId=value
```

Description

Specifies the node ID of an IceStorm replica, where *value* is a non-negative integer. The node ID is also used as the replica's priority, such that a larger value assigns higher priority to the replica. The replica with the highest priority becomes the coordinator of its group. This property must be defined for each replica.

service.Nodes.*id*

Synopsis

```
service.Nodes.id=value
```

Description

This property is used for a manual deployment of highly available IceStorm, in which each of the replicas must be explicitly configured with the proxies of all other replicas. The value is a proxy for the replica with the given node *id*. A replica's object identity has the form *instance-name*/node*id*, such as DemoIceStorm/node2.

service.Publish.*AdapterProperty*

Synopsis

service.Publish.*AdapterProperty*=*value*

Description

IceStorm uses the adapter name *service*.Publish for the object adapter that processes incoming requests from publishers. Therefore, adapter properties can be used to configure this adapter.

service.ReplicatedPublishEndpoints

Synopsis

service.ReplicatedPublishEndpoints=*value*

Description

This property is used for a manual deployment of highly available IceStorm. It specifies the set of endpoints returned for the publisher proxy returned from IceStorm::Topic::getPublisher.

If this property is not defined, the publisher proxy returned by a topic instance points directly at that replica and, should the replica become unavailable, publishers will not transparently failover to other replicas.

service.ReplicatedTopicManagerEndpoints

Synopsis

service.ReplicatedTopicManagerEndpoints=*value*

Description

This property is used for a manual deployment of highly available IceStorm. It specifies the set of endpoints used in proxies that refer to a replicated topic. This set of endpoints should contain the endpoints of each IceStorm replica.

For example, the operation IceStorm::TopicManager::create returns a proxy that contains this set of endpoints.

service.Send.Timeout

Synopsis

service.Send.Timeout=*num*

Description

IceStorm applies a send timeout when it forwards events to subscribers. The value of this property determines how long IceStorm will wait

for forwarding of an event to complete. If an event cannot be forwarded within *num* milliseconds, the subscriber is considered dead and its subscription is cancelled. The default value is 60 seconds. Setting this property to a negative value disables timeouts.

service.Send.QueueSizeMax

Synopsis

`service.Send.QueueSizeMax=num`

Description

The value of this property determines how many events can be queued for a subscriber by IceStorm. When the maximum size is reached, the old events will either be dropped or the subscriber will be removed. Setting this property to a negative value specifies an infinite queue size. The default value is -1.

service.Send.QueueSizeMaxPolicy

Synopsis

`service.Send.QueueSizePolicy=RemoveSubscriber|DropEvents`

Description

The value of this property specifies how IceStorm will behave if the maximum queue size is reached for a subscriber. If set to `RemoveSubscriber`, IceStorm will remove the subscriber as soon as the limit is reached. If set to `DropEvents`, older events will be removed to make room for new events. The default value is `RemoveSubscriber`.

service.TopicManager.*AdapterProperty*

Synopsis

`service.TopicManager.AdapterProperty=value`

Description

IceStorm uses the adapter name `service.TopicManager` for the topic manager's object adapter. Therefore, adapter properties can be used to configure this adapter.

service.Trace.Election

Synopsis

`service.Trace.Election=num`

Description

Trace activity related to elections:

0	No election trace (default).
1	Trace election activity.

service.Trace.Replication

Synopsis

`service.Trace.Replication=num`

Description

Trace activity related to replication:

0	No replication trace (default).
1	Trace replication activity.

service.Trace.Subscriber

Synopsis

`service.Trace.Subscriber=num`

Description

The subscriber trace level:

0	No subscriber trace (default).
1	Trace topic diagnostic information on subscription and unsubscription.
2	Like 1, but more verbose, including state transitions for a subscriber (such as going offline after a temporary network failure, and going online again after a successful retry, etc.).

service.Trace.Topic

Synopsis

`service.Trace.Topic=num`

Description

The topic trace level:

0	No topic trace (default).
1	Trace topic links, subscription, and unsubscription.
2	Like 1, but more verbose, including QoS information, and other diagnostic information.

service.Trace.TopicManager

Synopsis

service.Trace.TopicManager=*num*

Description

The topic manager trace level:

0	No topic manager trace (default).
1	Trace topic creation.

service.Transient

Synopsis

service.Transient=*num*

Description

If *num* is a value greater than zero, IceStorm runs in a fully transient mode in which no database is required. Replication is not supported in this mode. If not defined, the default value is zero.

IceStormAdmin.*

On this page:

IceStormAdmin.Host

Synopsis

`IceStormAdmin.Host=`*`host`*

Description

When used together with `IceStormAdmin.Port`, `icestormadmin` uses the `Finder` interface to discover the topic manager at the specified host and port.

> `icestormadmin` ignores this setting if you define one or more `IceStormAdmin.TopicManager` properties.

IceStormAdmin.Port

Synopsis

`IceStormAdmin.Port=`*`port`*

Description

When used together with `IceStormAdmin.Host`, `icestormadmin` uses the `Finder` interface to discover the topic manager at the specified host and port.

> `icestormadmin` ignores this setting if you define one or more `IceStormAdmin.TopicManager` properties.

IceStormAdmin.TopicManager.Default

Synopsis

`IceStormAdmin.TopicManager.Default=`*`proxy`*

Description

Defines the proxy for the default IceStorm topic manager. This property is used by `icestormadmin`. IceStorm applications may choose to use this property for their configuration as well.

IceStormAdmin.TopicManager.*name*

Synopsis

`IceStormAdmin.TopicManager.`*`name`*`=`*`proxy`*

Description

Defines a proxy for an IceStorm topic manager for `icestormadmin`. Properties with this pattern are used by `icestormadmin` if multiple topic managers are in use, for example:

```
IceStormAdmin.TopicManager.A=A/TopicManager:tcp -h x -p 9995
IceStormAdmin.TopicManager.B=Foo/TopicManager:tcp -h x -p 9995
IceStormAdmin.TopicManager.C=Bar/TopicManager:tcp -h x -p 9987
```

This sets the proxies for three topic managers. Note that *name* need not match the instance name of the corresponding topic manager — *name* simply serves as a tag. With these property settings, the `icestormadmin` commands that accept a topic can now specify a topic manager other than the default topic manager that is configured with `IceStormAdmin.TopicManager.Default`. For example:

```
current Foo
create myTopic
create Bar/myOtherTopic
```

This sets the current topic manager to the one with instance name `Foo`; the first `create` command then creates the topic within that topic manager, whereas the second `create` command uses the topic manager with instance name `Bar`.

Ice Release Notes

The release notes provide information about an Ice release, including descriptions of significant new features and changes, instructions for upgrading from an earlier release, and important platform-specific details.

Topics

Supported Platforms for Ice 3.7.1

Ice 3.7.1 is supported on the platform, compiler, and environment combinations shown below. Other platforms and compilers might work as well but have not been tested. Please contact us if you need support for a platform or compiler that is not on this list.

On this page:

Ice for C++

Run-Time Platform	Compiler	Run-Time Architecture	Development Platform
Windows 7 Windows 8.1 Windows 10 Windows Server 2012 Windows Server 2016	Visual Studio 2010, Visual Studio 2013, Visual Studio 2015, Visual Studio 2017	x86, x64	Same as Run-Time
Windows 10 UWP (Universal Windows)	Visual Studio 2015, Visual Studio 2017	x86, x64	Same as Run-Time
Red Hat Enterprise Linux 7 SUSE Linux Enterprise Server 12 Debian 9 (Stretch) Ubuntu 16.04 (Xenial Xerus) Ubuntu 18.04 (Bionic Beaver) Amazon Linux (amzn1)	GCC (default version) GCC 7.2	x86_64, x86 x86_64 amd64, armhf amd64 amd64 x86_64	Same as Run-Time
Linux Yocto 2.4 (Rocko)	Poky Yocto 2.4 (Rocko)	armhf	Ubuntu 16.04 amd64
macOS 10.12 (Sierra) and 10.13 (High Sierra)	Xcode 9	x86_64	Same as Run-Time
iOS 9, 10 and 11	Xcode 9	armv7, armv7s, arm64, iOS Simulator	macOS 10.12 and 10.13

> Ice for C++ provides two separate Slice to C++ mappings, the Slice to C++11 mapping and the Slice to C++98 mapping. You can use either (or both) mappings with all the C++ compilers listed above, with the exception of Visual Studio 2010 and Visual Studio 2013–Ice for C++ supports only the C++98 mapping with these older versions of Visual Studio.

Ice for C#/.NET

Platform	Compiler	Architecture
Windows 7 Windows 8.1 Windows 10 Windows Server 2012 Windows Server 2016	Visual Studio 2013, Visual Studio 2015, Visual Studio 2017 targeting .NET framework 4.5.1 or later	x86, x64

Windows 7 Windows 8.1 Windows 10 Windows Server 2012 Windows Server 2016 Red Hat Enterprise Linux 7 Debian 9 (Stretch) Ubuntu 16.04 (Xenial Xerus) Ubuntu 18.04 (Bionic Beaver)	.NET Core 2.1 SDK and .NET Core Runtime 2.0.6	x64 x86_64 amd64 amd64 amd64

Ice for Java

Platform	Environment
All Ice for C++ platforms	JDK 8, JDK 9 and JDK 10 JDK 7 (only with the Java Compat Mapping)
Android 7.0 to 8.0 with the Java Mapping	JDK 8, Android Studio 3.x
Android 5.0 to 8.0 with the Java Compat Mapping	JDK 8, Android Studio

Ice for JavaScript

Platform	Environment
Web browser with ECMAScript 5	Recent versions of Chrome, Edge, Firefox, Internet Explorer, Safari
Web browser with ECMAScript 6	Recent versions of Chrome, Edge, Firefox, Safari
Windows 7 Windows 8.1 Windows 10 Ubuntu 16.04 (Xenial Xerus) Ubuntu 18.04 (Bionic Beaver) macOS 10.12 (Sierra) and 10.13 (High Sierra)	Node.js 8.x or 9.x Node.js 4.2 or greater with the ES5 mapping

Ice for MATLAB

Platform	Environment	Architecture
Windows 7 Windows 8.1 Windows 10	MATLAB 2016a to 2018a	x64

Ice for Objective-C

Run-Time Platform	Compiler	Run-Time Architecture	Development Platform
macOS 10.12 (Sierra) and 10.13 (High Sierra)	Xcode 9	x86_64	Same as Run-Time
iOS 9, 10 and 11	Xcode 9	armv7s, arm64, iOS Simulator	macOS 10.12 (Sierra) and 10.13 (High Sierra)

Ice for PHP

Platform	Environment	Architecture
Windows 7 Windows 8.1 Windows 10 Windows Server 2012 Windows Server 2016	PHP 7.0 to 7.2	x86, x64
Red Hat Enterprise Linux 7 Amazon Linux SUSE Linux Enterprise Server 12 Debian 9 (Stretch) Ubuntu 16.04 (Xenial Xerus) Ubuntu 18.04 (Bionic Beaver)	PHP 5.4 PHP 5.3 PHP 5.5 PHP 7.0 PHP 7.0 PHP 7.2	x86, x86_64 x86_64 x86_64 amd64 amd64 amd64
macOS 10.12 (Sierra) and 10.13 (High Sierra)	PHP 7.2	x86_64

Ice for Python

Platform	Environment	Architecture
Windows 7 Windows 8.1 Windows 10 Windows Server 2012 Windows Server 2016	Python 2.7, Python 3.6	x86, x64
Red Hat Enterprise Linux 7 Amazon Linux SUSE Linux Enterprise Server 12 Debian 9 (Stretch) Ubuntu 16.04 (Xenial Xerus) Ubuntu 18.04 (Bionic Beaver) Linux Yocto 2.4 (Rocko)	Python 2.7, Python 3.5	x86, x86_64 x86_64 x86_64 amd64 amd64 amd64 armhf
macOS 10.12 (Sierra) and 10.13 (High Sierra)	Python 2.7, Python 3.6	x86_64

Ice for Ruby

Platform	Environment	Architecture
Red Hat Enterprise Linux 7 Amazon Linux SUSE Linux Enterprise Server 12 Debian 9 (Stretch) Ubuntu 16.04 (Xenial Xerus) Ubuntu 18.04 (Bionic Beaver) macOS 10.12 (Sierra) and 10.13 (High Sierra)	Ruby 2.0 to Ruby 2.5	x86, x86_64 x86_64 x86_64 amd64 amd64 amd64 x86_64

New Features in Ice 3.7

This page describes notable additions and improvements in Ice 3.7. For a detailed list of the changes in this release, please refer to the changelog in the source tree. Our upgrade guide documents the changes that may affect the operation of your applications or have an impact on your source code.

On this page:

New Features Introduced in Ice 3.7.1

This section describes the new features first introduced with Ice 3.7.1.

New MATLAB Mapping

Ice provides a new Slice to MATLAB mapping, which allows you to develop Ice clients using only MATLAB. For example:

MATLAB

```
function client(args)

    addpath('generated');
    if ~libisloaded('ice')
        loadlibrary('ice', @iceproto)
    end

    import Demo.*

    if nargin == 0
        args = {};
    end

    communicator = Ice.initialize(args);
    cleanup = onCleanup(@() communicator.destroy());

    hello =
Demo.HelloPrx.checkedCast(communicator.stringToProxy('hello:default -h
localhost -p 10000'));
    hello.sayHello();

end
```

See the MATLAB Mapping chapter and Using the MATLAB distribution for more information.

Support for .NET Core 2.0 on Linux and Windows

You can now use Ice for C#/.NET to build .NET Core 2.0 applications on Linux and Windows. See Building Ice Applications for .NET for additional details.

In Ice 3.7.0 and Ice 3.6, Ice for C#/.NET was only available for Windows applications built with the .NET Framework.

New `ice_fixed` Proxy Factory Method

A fixed proxy is a proxy bound to a connection, and you need to create such a proxy to send a request from a server to a client over a bidirectional connection (in this context the sender of the request is the server and the receiver is the client).

The new proxy factory method `ice_fixed` creates a fixed proxy bound to the connection given as parameter.

As of this release, the preferred way to pass a proxy to a callback object from a client to a server is as a proxy. The server then calls `ice_fixed` on this proxy with `current.con` to create the fixed proxy it needs. Ice demo programs, such as `Ice/bidir`, use this pattern. In previous releases, the recommended approach was to send the identity of the client's callback object to the server; the server then created the fixed proxy by calling `createProxy` on the connection. Passing the callback proxy as a proxy instead of an identity is type-safe and follows our general recommendation to pass proxies as proxies - not as strings or identities.

Main New Features Introduced in Ice 3.7.0

This section describes the major new features introduced in Ice 3.7.0.

Ice-E and Ice Touch Merged into Ice

Ice-E and Ice Touch are now part of Ice. They are no longer separate products.

In particular, the latest Ice binary distribution for macOS includes support for Xcode SDKs and iOS. Refer to Using the macOS binary distribution for information on how to use the Xcode SDKs provided with the Ice binary distribution.

New C++11 Mapping

Ice now provides two distinct Slice to C++ mappings:

- The C++98 mapping, which corresponds to the C++ mapping provided by prior Ice releases.
- A new C++11 mapping, which takes full advantage of C++11 features, including standard smart pointers, move semantics, lambda expressions, futures and promises, and much more.

We provide an overview of some of the new features in the C++11 mapping below. Please refer to the C++11 Mapping and C++98 Mapping chapters for additional information.

Standard shared_ptr for Everything

With the C++11 mapping, virtually all Ice objects are manipulated through `std::shared_ptr` smart pointers. For example:

C++

```
// ich.communicator() is a std::shared_ptr<Ice::Communicator>
// oa is a std::shared_ptr<Ice::ObjectAdapter>
//
Ice::CommunicatorHolder ich(argc, argv);
auto oa = ich.communicator()->createObjectAdapter("Hello");

// servant is a std::shared_ptr<HelloI>
// proxy is a std::shared_ptr<Ice::ObjectPrx>
//
auto servant = make_shared<HelloI>();
auto proxy = oa->addWithUUID(servant);
oa->activate();

// address is a std::shared<Address> (Address is a mapped Slice class)
//
auto address = make_shared<Address>();
person->address = address;
```

AMI

With the C++11 mapping, Ice provides two options for asynchronous method invocation (AMI): a function that returns a `std::future` of the result, and a function that takes `std::function` "callbacks" to process the result. For example, the Slice operation `string getName()` is mapped to a proxy class with the following Async functions:

C++

```
std::future<std::string> getNameAsync(const Ice::Context& context =
Ice::noExplicitContext);

std::function<void()>
getNameAsync(std::function<void(std::string)> response,
             std::function<void(std::exception_ptr)> ex = nullptr,
             std::function<void(bool)> sent = nullptr,
             const Ice::Context& context = Ice::noExplicitContext);
```

Your code can then use the standard future API, or lambda expressions, to process the result of these asynchronous calls. For example:

C++ with std::future

```
auto fut = proxy->getNameAsync();

try
{
    cout << fut.get() << endl;
}
catch(const std::exception& ex)
{
   // something went wrong...
}
```

or

C++ with callbacks

```
proxy->getNameAsync([](string name) { cout << name << endl; },
                    [](std::exception_ptr eptr) { ... deal with error...
});
```

AMD

With the C++11 mapping, the API for asynchronous method dispatch (AMD) closely resembles its AMI counterpart with callbacks:

C++

```
virtual void getNameAsync(std::function<void(const std::string&)>
response,
                          std::function<void(std::exception_ptr)> ex,
                          const Ice::Current&) = 0;
```

Even though the response callbacks are not identical in AMI and AMD, they are compatible, which allows you to easily chain AMD and AMI calls:

C++

```
// AMD implementation of Person::getName
void
PersonI::getNameAsync(std::function<void(const std::string&)> response,
                      std::function<void(std::exception_ptr)> ex,
                      const Ice::Current&)
{
     // Call the same operation on _proxy with AMI
    _proxy->getNameAsync(response, ex);
}
```

Movable Parameters

On the server-side, the generated code allocates `in` parameters on the stack before dispatching the call to your operation implementation. With the C++98 mapping, "big" parameters such as strings, structs and sequences are passed as `const&`, just like on the client-side. With the C++11 mapping, they are passed as values, which allows you to move them. For example:

Slice

```
interface TextTransfer
{
    void sendText(string longText);
}
```

is mapped to:

C++

```
// Server-side
virtual void sendText(std::string, const Ice::Current&) = 0; //
std::string, not const std::string&
```

So you can now keep this parameter without additional memory allocation:

C++

```
// Implement sendText
void
TestTransferI::sendText(std::string longText, const Ice::Current&)
{
    std::lock_guard<std::mutex> lk(_mutex);
    _text = std::move(longText); // move-assignment
}
```

The same rule applies for any parameter that Ice allocates and then gives to your application, like (for example) return and out parameters given to AMI response callbacks.

New Java Mapping

Similar to what we've done for C++, Ice 3.7 supports two Java mappings: Java and Java Compat. As its name suggests, the Java Compat mapping is provided primarily for backward compatibility purposes so that existing Java applications can be upgraded to Ice 3.7 without requiring much change. Note however that the Java Compat mapping will be removed in the next release and we recommend migrating applications to the new Java mapping as soon as possible.

The primary goals of the new Java mapping were modernization, standardization, and simplification. We provide an overview of the new features below. Please refer to the Java Mapping and Java Compat Mapping chapters for additional information.

Regardless of whether you use the Java Compat mapping or the Java mapping, the minimum required Java version for Ice 3.7 is Java 8.

AMI

The API for asynchronous method invocation (AMI) now uses Java's `CompletableFuture` class. For example, the Slice operation `string getName()` would be mapped as follows:

Java

```
java.util.concurrent.CompletableFuture<String> getNameAsync()
```

Clients can easily use lambdas as asynchronous callbacks by configuring the future:

Java

```
proxy.getNameAsync().whenComplete((name, ex) ->
    {
        if(ex != null)
        {
            // oops!
        }
        else
        {
            System.out.println("got name " + name);
        }
    });
```

The `CompletableFuture` class provides a great deal of flexibility and power for implementing your asynchronous programming requirements.

AMD

As with AMI, the API for asynchronous method dispatch (AMD) has also changed significantly. Now the mapping closely resembles its AMI counterpart in that an AMD operation returns a `CompletionStage`:

Java

```
java.util.concurrent.CompletionStage<String>
getNameAsync(com.zeroc.Ice.Current current)
```

The implementation is responsible for creating and returning a completion stage that must eventually be completed with a result or an

exception. Typically the implementation will return an instance of `CompletableFuture`, which implements the `CompletionStage` interface.

Out Parameters

The Java mapping no longer uses "holder" classes to provide the values of out parameters. For Slice operations that return a single value (either as the return type or as an out parameter), the mapped method provides the value as its return type. Consider these operations:

Slice

```
interface Example
{
    string getNameRet();
    void getNameOut(out string s);
}
```

The mapping for these two operations is identical:

Java

```
public interface ExamplePrx ...
{
    String getNameRet();
    String getNameOut();
}
```

For Slice operations that return multiple values, the mapping generates an extra class to hold the operation's results:

Slice

```
interface Example
{
    string getAll(out int count);
}
```

The mapping for `getAll` is shown below:

Java

```
public interface Example ...
{
    public static class GetAllResult
    {
        ...
        public String returnValue;
        public int count;
    }
    ...
}

public interface ExamplePrx ...
{
    Example.GetAllResult getAll();
}
```

Changing the API to ensure that all mapped operations return at most one logical value enabled us to use `CompletableFuture` and `CompletionStage` in our AMI and AMD mappings, respectively.

Optional Values

As part of the standardization goal, optional parameters and data members are now mapped to the `java.util.Optional` family of classes.

Servant Interfaces and Tie Classes

Tie classes are no longer necessary or supported with the Java mapping. Tie classes were a useful implementation technique in prior Ice versions that allowed a servant to implement Slice operations without the need to extend a generated class, making it possible for the servant to extend an arbitrary base class.

Now all servant code is generated in an interface, so your servant class only needs to implement the generated interface and can extend an arbitrary base class without the need for a tie class.

Packaging

All classes have been moved to the `com.zeroc` package (e.g., `com.zeroc.Ice.Communicator`).

C# Support for async and await

The C# mappings for asynchronous method invocation (AMI) and asynchronous method dispatch (AMD) are completely new in Ice 3.7 and take advantage of the new asynchronous programming features introduced with .NET 4.5.

The `begin_/end_` proxy API from previous Ice versions is still supported for backward compatibility purposes but is now deprecated and will be removed in the next Ice version. No backward-compatible API is provided for AMD operations.

The sections below provide an overview of these changes.

AMI

The API for asynchronous method invocation (AMI) now uses .NET's `Task` class. For example, the Slice operation `string getName()` wo

uld be mapped as follows:

C#

```
System.Threading.Tasks.Task<string> getNameAsync()
```

Clients can use the `await` keyword to suspend processing of the current thread and resume it as a continuation when the task is complete:

C#

```
var name = await proxy.getNameAsync();

// use the return value in a continuation...
```

You could also interact with the task directly:

C#

```
var task = proxy.getNameAsync();

// do something else

var name = task.Result; // blocks until the result is available
```

Or configure a lambda to execute as the continuation:

C#

```
proxy.getNameAsync().ContinueWith((name) =>
    {
        Console.WriteLine("got name " + name);
    });
```

AMD

As with AMI, the API for asynchronous method dispatch (AMD) has also changed significantly. Now the mapping closely resembles its AMI counterpart in that an AMD operation returns a `Task`:

C#

```
System.Threading.Tasks.Task<string> getNameAsync(Ice.Current current)
```

The implementation is responsible for creating and returning a task that must eventually be completed with a result or an exception.

Out Parameters

The C# mapping for AMI and AMD no longer uses out parameters. For Slice operations that return a single value (either as the return type or

as an out parameter), the mapped method provides the value as its return type. Consider these operations:

Slice

```
interface Example
{
    string getNameRet();
    void getNameOut(out string s);
}
```

The AMI mapping for these two operations is identical:

C#

```
public interface ExamplePrx ...
{
    Task<string> getNameRetAsync();
    Task<string> getNameOutAsync();
}
```

For Slice operations that return multiple values, the mapping generates an extra type to hold the operation's results:

Slice

```
interface Example
{
    string getAll(out int count);
}
```

The mapping for `getAll` is shown below:

C#

```
public struct Example_GetAllResult
{
    ...
    public string returnValue;
    public int count;
}

public interface ExamplePrx ...
{
    Task<Example_GetAllResult> getAllAsync();
}
```

Changing the API to ensure that all mapped operations return at most one logical value enabled us to use tasks in our AMI and AMD mappings, respectively.

JavaScript

JavaScript 6 Updates

The JavaScript mapping has been updated to take advantage of the latest features present in the EcmaScript 6 standard:

- All Promise objects returned by the Ice run time are derived from the standard Promise type.
- The mapping for dictionaries is now based on the standard Map type. The `Ice.HashMap` type used in previous releases is still supported for a few cases that are not covered by the standard Map type. See JavaScript mapping for dictionaries for complete details.
- There is a new mapping for Slice modules based on the new JavaScript import and export keywords. See JavaScript mapping for modules for more details.

JavaScript Changes for AMD

The API for Asynchronous Method Dispatch (AMD) has been updated to use the standard Promise class. There are several differences with respect to the previous API:

- The `["amd"]` metadata is ignored by the Slice to JavaScript compiler.
- The Servant skeleton does not generate separate `_async` methods for AMD.
- At run time a servant can take advantage of AMD by returning a Promise object.

Consider the following example, where we define a `getName` operation in Slice:

```
module M
{
    interface Example
    {
        string getName();
    }
}
```

The JavaScript implementation can decide at run time to use the synchronous or asynchronous dispatch. If the servant returns a Promise, the Ice run time uses asynchronous dispatch and marshals the result when the promise is resolved. Otherwise, Ice uses synchronous dispatch and marshals the result right away. Here's an example that demonstrates the mapping:

```
class ServantI extends Servant
{
    getName(current)
    {
        if(_name !== undefined)
        {
            return _name; // Use synchronous dispatch
        }
        else
        {
            // Use asynchronous dispatch.
   // All JavaSript proxy invocations return a standard Promise object.
            // We can directly return the promise returned by getName
from our
            // servant implementation.
            return _proxy.getName();
        }
    }
}
```

Python Changes for AMI and AMD

We've added a new AMI mapping and modified the AMD mapping in Python.

> The existing AMI mapping is still supported for backward compatibility, but new applications should use the new AMI mapping. The changes to the AMD mapping break backward compatibility and require modifications to existing applications. Refer to the upgrade guide for more information.

AMI

The new API for asynchronous method invocation (AMI) uses future objects and adds the `Async` suffix to proxy methods. For example, the Slice operation `string getName()` would be mapped as follows:

Python

```
def getNameAsync()
```

Asynchronous proxy methods return instances of `Ice.Future`, which has an API similar to the built-in Python types `asyncio.Future` and `concurrent.futures.Future`.

Clients can easily configure the future with a completion callback:

Python

```
def callback(future):
    try:
        print("got name " + future.result())
    except:
        # oops!

proxy.getNameAsync().add_done_callback(callback)
```

The `Ice.Future` class also provides methods for checking the status of an invocation, cancelling an invocation, and blocking until the invocation completes, among others.

AMD

The API for asynchronous method dispatch (AMD) has changed significantly:

- The `_async` suffix is no longer used in the name of a dispatch method
- The dispatch method does not receive a callback object
- The dispatch method can either return results directly (like a synchronous dispatch method), or return a future object to be completed later

In essence, the synchronous and asynchronous mappings have been merged into a single mapping in which the value returned at run time dictates the semantics of the dispatch. If the implementation returns a future, Ice configures it with a callback and marshals the results when the future completes. For all other return types, Ice assumes the dispatch completed successfully and marshals the results immediately.

Here's an example:

Python

```
def getName(self, current=None):
    if self._name:
        return self._name # Name was cached, synchronous dispatch
    else:
        f = Ice.Future()  # We have to complete this future eventually
        ...
        return f          # Asynchronous dispatch
```

Python 3.5 Features

The Python mapping adds the following features for applications using Python 3.5 or later:

- Servant dispatch methods can be implemented as coroutines
- `Ice.Future` objects are awaitable, meaning they can be targets of the `await` keyword

Aside from these features, all of the improvements to the Python mapping can be used in Python 2.x or later.

New Database Back-end for IceGrid and IceStorm

IceGrid and IceStorm now store their data in LMDB databases. LMDB is a popular embedded database system known for its speed and reliability.

In prior Ice releases, IceGrid and IceStorm relied on Freeze for persistent storage, and Freeze itself stores its data in Oracle Berkeley DB databases. The migration instructions describe how to migrate your existing IceGrid and IceStorm databases to the new LMDB format.

Bluetooth Transport for Linux and Android

Our newest transport plug-in enables Ice applications on Linux and Android to communicate with one another via Bluetooth. Once two devices are paired, establishing a peer-to-peer Bluetooth connection is just as convenient as with any other Ice transport. For example, a client can use a stringified proxy like this:

```
objectId:bt -a "01:23:45:67:89:AB" -u dfde1c02-d907-4ca1-bd99-31804c569624
```

The `-a` option denotes the device address of the peer and the `-u` option defines the unique identifier (UUID) associated with the desired Ice service on the peer device. It's also possible to secure the bluetooth connection with SSL by configuring IceSSL and using a `bts` endpoint instead.

Our ice-demos repository includes a new Android sample program that demonstrates how to use the transport. Refer to the Ice manual for more information on the plug-in.

iAP Transport for iOS

The new Ice iAP transport enables iOS clients to communicate with accessories over Bluetooth or the Apple Lightning or 30-pin connector.

> This transport requires special support on the accessory-side and is only provided as a preview to demonstrate how it can be used in iOS applications to support Bluetooth communications. Please contact us for more information.

For example, a client can use a stringified proxy like this to communicate with an accessory that advertises the `com.zeroc.helloWorld` pr otocol:

```
objectId:iap -p com.zeroc.helloWorld
```

It's also possible to secure the accessory connection with SSL by configuring IceSSL and using an `iaps` endpoint instead.

Refer to the Ice manual for more information on the plug-in.

Other New Features Introduced in Ice 3.7.0

This section describes other noteworthy features introduced in Ice 3.7.0

Optional Semicolons after Braces in Slice

Semicolons are now optional after a closing brace in Slice. For example, with Ice releases up to Ice 3.6, you would write:

Slice

```
module M
{
    enum E { A, B, C, D };

    interface Foo
    {
        void op();
    };
};
```

With Ice 3.7, the definitions above remain valid, but you can also remove the semicolons after the closing braces, as in:

Slice

```
module M
{
    enum E { A, B, C, D }

    interface Foo
    {
        void op();
    }
}
```

Simplified Communicator Destruction

Ice 3.7 provides support for automatic communicator destruction in most programming languages.

- C++: `Ice::CommunicatorHolder` RAII Helper
- C#: Communicator implements `IDisposable` (since Ice 3.6)
- Java: Communicator is `AutoCloseable`
- Python: Communicator implements the Python context manager protocol

Freeze Unbundled

The Freeze persistent service is no longer included in the ice source repository or in the Ice binary distributions. It's now a separate add-on, with its own source repository and binary distributions.

Backward Compatibility of Ice Versions

The subsections below describe how Ice maintains (or does not maintain) backwards compatibility between versions.

On this page:

Source-code compatibility

Ice maintains source-code compatibility between a patch release (e.g., 3.7.1) and the most recent minor release (e.g., 3.7.0), but does not guarantee source-code compatibility between minor releases (e.g., between 3.6 and 3.7).

Upgrading your Application from Ice 3.6 describes the significant API changes in this release that may impact source-code compatibility. Furthermore, the subsections Removed APIs and Deprecated APIs summarize additional changes to Ice APIs that could affect your application.

Binary compatibility

As for source-code compatibility, Ice maintains backward binary compatibility between a patch release and the most recent minor release, but does not guarantee binary compatibility between minor releases.

The requirements for upgrading to a new minor (or major) release depend on the language mapping used by your application:

- For statically-typed languages (C++, Java, .NET), the application must be recompiled.
- For scripting languages that use static translation, your Slice files must be recompiled.
- No action is necessary for a Python or Ruby script that loads its Slice files dynamically.

On-the-wire compatibility

Ice always maintains "on the wire" compatibility with prior releases. A client using Ice version *x* can communicate with a server using Ice version *y* and vice versa.

The Ice message format is governed by two sets of rules: the protocol rules and the encoding rules. The protocol rules define the number of message types as well as the format of the message headers. The encoding rules specify how Slice types are marshaled. Ice maintains separate versions for the protocol and encoding, which makes it possible for them to evolve independently.

Several features introduced in Ice 3.5 required changes to the way that Slice classes and exceptions are marshaled. These changes make the encoding incompatible with previous releases, and therefore Ice 3.5 introduced version 1.1 of the Ice encoding. No additional encoding changes were made in Ice 3.6 and Ice 3.7.

The protocol remains the same since the first version Ice, with protocol version number 1.0.

Changing the protocol or encoding format can be disruptive and does not happen often. Naturally, these changes raise the issue of backward compatibility with applications that use earlier versions of Ice, and more specifically, applications that use version 1.0 of the Ice encoding. Ice 3.5, 3.6 and 3.7 use the following semantics:

- Encoding version 1.1 is the default for these releases. You can change this default using the `Ice.Default.EncodingVersion` property.
- A proxy is marshaled with an embedded encoding version representing the highest version supported by the target object, so the receiver of a proxy always knows the encoding version(s) it is allowed to use when invoking operations on that proxy. A proxy created locally uses the default encoding version, which can be inspected and changed using proxy methods.
- The results of an operation are encoded using the same version as the incoming request.
- As you would expect, the features provided by the 1.1 encoding are unavailable when an Ice application is forced to use the 1.0 encoding.

See Encoding Version 1.1 for additional details.

Interface compatibility

Although Ice always maintains compatibility at the protocol level, changing Slice definitions can also lead to incompatibilities. As a result, Ice maintains interface compatibility between a patch release and the most recent minor release, but does not guarantee compatibility between minor releases.

This issue is particularly relevant if your application uses Ice services such as IceGrid or IceStorm, as a change to an interface in one of these services may adversely affect your application.

Interface changes in an Ice service can also impact compatibility with its administrative tools, which means it may not be possible to administer an Ice 3.7 service using a tool from a previous minor release (or vice-versa).

IceGrid

Starting with Ice 3.2, IceGrid registries and nodes are interface-compatible so you can use different IceGrid registry and node versions within the same deployment. For example, you can use an IceGrid node from Ice 3.2 with a registry from Ice 3.7.

IceGrid master and slaves can have different versions as long as they use the same database format. Since the IceGrid database format changed in Ice 3.6, an IceGrid slave from earlier versions will not be able to synchronize with an IceGrid master from Ice 3.6 or later. You need to upgrade all your IceGrid registries to Ice 3.6 or 3.7 to ensure replication between the registries work.

An IceGrid node can activate Ice servers that use different Ice releases, for example, an IceGrid node version 3.7 can activate an Ice 3.4 server.

The IceGrid graphical and command-line administrative tools can only administer an IceGrid registry with the same `major.minor` version. For example, you have to use the 3.7 administrative tools to administer an IceGrid 3.7 registry; you cannot use an administrative tool from an earlier Ice version. The reverse is also true: you cannot administer an IceGrid 3.6 registry with an IceGrid 3.7 administration tool.

IceStorm

Topic linking is supported between all IceStorm versions released after 3.0.

Upgrading your Application from Ice 3.7.x

Ice 3.7.1 maintains full source and binary compatibility with Ice 3.7.0: you can upgrade your application from Ice 3.7.0 to Ice 3.7.1 without recompiling or relinking anything.

The database formats used by Ice services such as IceGrid and IceStorm have not changed, therefore no database migration is required. Generally speaking, you are free to use any combination of Ice 3.7.x applications and Ice services.

> Some internal Ice interfaces may change in a patch release such as Ice 3.7.1, and therefore you must always upgrade all the Ice components of a given application together. For example, you should not use version 3.7.1 of the `ice37.dll` with version 3.7.0 of the `icessl37.dll` in the same application.

Upgrading your Application from Ice 3.6

The subsections below provide additional information about upgrading to Ice 3.7, including administrative procedures for the supported platforms.

On this page:

C++ Changes

You should be able to rebuild your C++ source code with Ice 3.7 with few if any changes, provided you use the Ice C++98 mapping of Ice 3.7. Upgrading to the new Ice C++11 mapping is a bigger undertaking which requires extensive changes to your source code. As a result, this section deals only with upgrades to Ice 3.7 with the C++98 mapping.

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

IceUtil Library Removed

The core Ice libraries in Ice 3.6 and prior releases were Ice and IceUtil. As of Ice 3.6, IceUtil was merged into Ice, so you can no longer link with IceUtil. On Linux and macOS, you need to remove `-lIceUtil` from your Makefiles. On Windows, you probably don't need to do anything since all Ice libraries are linked through `pragma comment lib` directives in header files (this linking through header files was introduced in Ice 3.6.0).

Stream API

We made significant changes to the Streaming interfaces in all language mappings. In C++, `InputStream` and `OutputStream` are now created through their constructors and typically stack allocated; in previous releases, they were heap-allocated and created using factory functions in the `Ice` namespace. Please refer to C++ Streaming Interfaces for complete details.

The command-line option `--stream` is no longer supported by `slice2cpp`.

Dispatch Interceptors

The dispatch and ice_dispatch functions have changed slightly: they now return a bool that indicates whether the request was dispatched synchronously (true) or asynchronously with AMD (false). They previously returned an enumerator.

IceSSL Certificate Creation

IceSSL Certificates are now created through factory functions, such as `cert = IceSSL::Certificate::load("myCert.pem")`. In previous releases, you would create them directly with the constructors of the `Certificate` class.

IceSSL Connection Info

The `certs` member of `IceSSL::ConnectionInfo` class is now a sequence of native certificate objects; in previous releases it was a sequence of string elements containing the PEM encoded certificates. The `IceSSL::NativeConnectionInfo` type that used to provide the native certificates has been removed.

OpenSSL Context with IceSSL

The member functions `setContext` and `getContext`, used to set or retrieve an OpenSSL context, are now on the `IceSSL::OpenSSL::Plugin` class.

C++11 Extensions

Ice 3.6 provided a few extensions for C++11-capable compilers, primarily additional AMI overloads with `std::function` parameters (suitable for lambda expression arguments). These extensions are no longer included in the C++98 mapping. If you want to take advantage of the C++11 features provided by your C++ compiler, you should upgrade to the Ice C++11 mapping.

`lib/c++11` and ++11 Libraries on Linux

The Ice 3.6 binary distributions for Linux include libraries with a ++11 suffix, such as `libIce++11.so.36`, and a `c++11` subdirectory in `lib` or `lib64` with symbolic links to these ++11 libraries. These libraries correspond to a build of Ice using GCC with `--std=c++11` turned on.

This "C++11 build" of Ice 3.6 provides some C++11 extensions available only in C++11 mode (see above).

In Ice 3.7, these ++11 libraries correspond to the Ice C++11 mapping, and there is no longer a `c++11` subdirectory of `lib`. With Ice 3.7 and GCC 4.8.2 or greater, you should be able to use the Ice C++98 binaries in any mode (`default`, `--std=c++11`, etc.). See the GCC Cxx11 AbiCompatibility page for more information on ABI compatibility with GCC. Symbolic links to these C++98 libraries (`libIce.so`, `libIceGrid.so`, etc.) are in the main `lib` directory.

C# Changes

You should be able to rebuild your C# source code with Ice 3.7 with few if any changes.

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

AMD

The AMD (asynchronous method dispatch) mapping has been updated to be Task-based. Applications using AMD should update their servant implementations to use the new Task-based mapping. Refer to the Asynchronous Method Dispatch (AMD) in C-Sharp page for details of the new AMD mapping.

Stream API

We made significant changes to the Streaming interfaces in all language mappings. In C#, `InputStream` and `OutputStream` are now created through their constructors; in previous releases, they were created using factory functions in the `Ice.Util` class. Please refer to C-Sharp Streaming Interfaces for complete details.

The command-line option `--stream` is no longer supported by `slice2cs`.

Dispatch Interceptors

The dispatch and ice_dispatch functions have changed slightly: they now return a `Task<Ice.OutputStream>` for asynchronous dispatch or null otherwise. They previously returned an enumerator.

IceSSL Connection Info

The `certs` member of `IceSSL::ConnectionInfo` class is now a sequence of native certificate objects; in previous releases it was a sequence of string elements containing the PEM encoded certificates. The `IceSSL.NativeConnectionInfo` type that used to provide the native certificates has been removed.

Java Changes

You should be able to rebuild your Java source code with Ice 3.7 with few if any changes, provided you use the Java Compat mapping of Ice 3.7. Upgrading to the new Java mapping is a bigger undertaking which requires extensive changes to your source code. As a result, this section deals only with upgrades to Ice 3.7 with the Java Compat mapping.

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

New `slice2java` Option

To use the Java Compat mapping, you must add the new `--compat` option to your invocations of `slice2java`. For Gradle projects, you can set the property `slice.compat = true` to enable the Java Compat mapping.

JAR Filenames

The names of the JAR files for the Java Compat run time now include `compat`, such as `ice-compat-3.7.0.jar`.

`IceUtil` Package Removed

The following classes have moved to the Freeze repository:

- `IceUtil.Cache`
- `IceUtil.FileLockException`
- `IceUtil.Store`

Stream API

We made significant changes to the Streaming interfaces in all language mappings. In Java, `InputStream` and `OutputStream` are now created through their constructors; in previous releases, they were created using factory functions in the `Ice.Util` class. Please refer to Java Streaming Interfaces for complete details.

The command-line option `--stream` is no longer supported by `slice2java`.

Dispatch Interceptors

The dispatch and ice_dispatch functions have changed slightly: they now return a bool that indicates whether the request was dispatched synchronously (true) or asynchronously with AMD (false). They previously returned an enumerator.

IceSSL Connection Info

The `certs` member of `IceSSL::ConnectionInfo` class is now a sequence of native certificate objects; in previous releases it was a sequence of string elements containing the PEM encoded certificates. The `IceSSL.NativeConnectionInfo` type that used to provide the native certificates has been removed.

JavaScript Changes

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

Class Helpers

The `Ice.Class` helper method has been removed. The JavaScript `class` keyword or a third-party helper should be used to declare JavaScript classes.

Dictionary Mapping

The mapping for dictionaries has been updated to use the standard JavaScript `Map` type when possible. `Ice.HashMap` is still used for dictionaries with mutable keys and its API has been updated to match that of JavaScript Map.

Promise Usage

The `Ice.Promise` class in previous version was a custom implementation of the Promise/A+ specification. It has been updated to be an extension of the standard JavaScript Promise type and most of the non-standard methods have been removed:

- `Ice.Promise.prototype.exception` method has been removed, use Promise.prototype.catch instead.
- `Ice.Promise.prototype.succeed` has been removed, use Promise.prototype.resolve instead. The `succeed` method accepted

a variable number of arguments; with `resolve` you can achieve the same by passing an array with the values.
- `Ice.Promise.prototype.fail` has been removed, use Promise.prototype.reject instead. The `fail` method accepted a variable number of arguments; with `reject` you can achieve the same by passing an array with the values.
- `Ice.Promise.prototype.succeeded`, `Ice.Promise.prototype.failed` and `Ice.Promise.prototype.completed` methods have been removed and there are replacements in the standard Promise type. These methods were rarely used in practice.
- `Ice.Promise` completion callbacks no longer provide an `Ice.AsyncResult` parameter as the last argument. If you need to use it you must keep a reference to it when invoking a method.
- `Ice.Promise.all` has been removed, use Promise.all instead.

AMD

The ["amd"] metadata is ignored by the `slice2js` compiler. The compiler no longer generates a separate method that receives an AMD callback with `ice_response` and `ice_exception` member methods. Instead a method can take advantage of AMD (asynchronous method dispatch) by returning a Promise object from a servant method.

Mapping for Sequence of Bytes

The mapping for `sequence<byte>` is always the `Uint8Array` JavaScript type; previously the NodeJS engine used a NodeJS `Buffer` type and browser engines used a `Uint8Array`. The helper method `Ice.Buffer.createNative` has been removed; the `Uint8Array` constructor should be used instead.

Objective-C Changes

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

Dispatch Interceptors

The dispatch and ice_dispatch functions have changed slightly: they no longer return a bool, user exceptions are now raised by `ice_dispatch`.

ICEInputStream

The `wrapInputStream` method has been removed from the `ICEUtil` class.

PHP changes

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

Namespace Usage

The Ice for PHP extension included in binary distributions is now built with namespaces enabled. All Ice definitions are placed inside Ice namespaces, and the default mapping for a Slice module is a PHP namespace with the same name. The old flattened mapping has been deprecated and will be removed in a future release.

To use the old flattened mapping, you need a custom build of the Ice for PHP extension with namespaces disabled and you need to pass the `--no-namespaces` option to `slice2php` when compiling your Slice files. Consult the PHP build instructions for details of how to build the PHP extension with namespaces disabled.

Loading Ice

The Ice run time is loaded by `require Ice.php` independently of whether you are using the namespace mapping (default) or the flattened mapping (deprecated). Previously, applications using the namespace mapping needed to load `Ice_ns.php`.

Ice Unset

The unset value for optional parameters with the namespace mapping is `\Ice\None` rather than `\Ice\Unset`; the latter cannot be used as `unset` is a PHP keyword. `Ice_Unset` is still available with the flattened mapping.

Python Changes

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

AMD

The mapping for asynchronous method dispatch (AMD) has changed significantly. Asynchronous dispatch methods in existing applications will need to be modified as follows:

- Remove the `_async` suffix from the method name
- Remove the callback parameter
- Change how the method reports results and exceptions
- Return an `Ice.Future` object

A dispatch method now has the option of using synchronous semantics or asynchronous semantics. It can return results directly, in which case Ice marshals the results immediately, or it can return a future that the implementation must complete later.

Calls to `ice_response` on the callback object must be converted to calls to `set_result` on the future object. Similarly, calls to `ice_exception` on the callback object must be converted to calls to `set_exception` on the future object.

Consider this operation:

Slice

```
string getResults(int id, out bool validated);
```

Suppose we have this existing implementation:

Python

```
def getResults_async(self, cb, id, current=None):
    cb.ice_response("answer", True) # Typically done later, e.g., in a
separate thread
```

Using Ice 3.7, we need to convert this implementation as follows:

Python

```
def getResults(self, id, current=None): # Changed method name, removed
callback parameter
    f = Ice.Future()
    f.set_result(("answer", True))      # Typically done later, e.g., in
a separate thread
    return f                            # Return a future
```

Pay special attention to the value passed to `set_result`: this method accepts only a single value. If the operation returns multiple values, they must be supplied in a tuple.

Ruby Changes

> See also the subsections Changed APIs and Removed APIs for information about APIs defined in Slice and therefore common to all language mappings.

Freeze Persistence Service

The Freeze persistence service, which allows you to store objects defined in Slice in a Berkeley DB database, is no longer bundled with Ice. It is now a separate add-on.

Migrating the IceGrid and IceStorm Databases from Freeze to LMDB

As of Ice 3.7, IceGrid and IceStorm rely on LMDB for persistent storage. In prior releases, IceGrid and IceStorm were using the Freeze service for persistent storage; Freeze itself stores its data in Oracle Berkeley DB.

Berkeley DB and LMDB are quite similar: they are both embedded database libraries that require little or no administration and configuration. They both maintain persistent key-value maps, where keys and values are sequences of bytes. While Berkeley DB creates many files in its DB environment (one file for each persistent map, log files and more), LMDB creates just two files in its own database enviromnent: a data file (data.mdb) that contains all the persistent maps, and a lock file (lock.mdb). There are no log files with LMDB, which further simplifies administration compared to Berkeley DB.

This section describes how to migrate an IceGrid registry or an IceStorm instance using Ice 3.5 or 3.6 (with a Freeze database) to an IceGrid registry or IceStorm instance using Ice 3.7 (with a LMDB database).

IceGrid Migration

Prerequisite: you need the IceGrid database export tool version 3.5 (`icegriddb35`) or version 3.6 (`icegriddb`). This utility is included in the Ice 3.6 distribution starting with Ice 3.6.2, but was not included in any Ice 3.5 distribution. If you are migrating from Ice 3.5, you need to build this export tool from sources: icegriddb35.

To start this migration, first stop the IceGrid registry you wish to upgrade, then export the Freeze/Berkeley DB database environment of your IceGrid registry:

With icegriddb 3.5 or 3.6

```
icegriddb --export icegriddb.ixp --dbhome /var/icegrid/db
```

The `icegriddb` export tool's version must match the existing IceGrid registry's version; for example, use `icegriddb35` with an IceGrid registry 3.5.x.

The resulting file (`icegriddb.ixp` in our example) is a binary file with the full content of the IceGrid registry database. The `icegriddb` utility can import this file into a Freeze/Berkeley DB or LMDB database.

Next, create a directory for your new IceGrid registry LMDB database files:

```
mkdir /var/icegrid/lmdb
```

Next, import `icegriddb.ixp` into this new LMDB database environment directory:

With icegriddb 3.7 or greater

```
icegriddb --import icegriddb.ixp --dbpath /var/icegrid/lmdb
```

Finally, edit your IceGrid registry configuration to replace the `IceGrid.Registry.Data` property with the new `IceGrid.Registry.LMDB.Path` property:

```
IceGrid.Registry.LMDB.Path=/var/icegrid/lmdb
```

> While the instructions above are sufficient for most deployments, you may want to review IceGrid Persistent Data and IceGrid Database Utility for detailed information about the tool and LMDB configuration options.

If you are upgrading the master IceGrid registry in a replicated environment and the slaves are still running, you should first restart the master registry in read-only mode using the `--readonly` option, for example:

```
icegridregistry --Ice.Config=config.master --readonly
```

Next, you can connect to the master registry with `icegridadmin` or the IceGrid administrative GUI from Ice 3.7 to ensure that the database is correct. If everything looks fine, you can shutdown and restart the master registry without the `--readonly` option.

IceGrid slaves from Ice <= 3.5 won't interoperate with the IceGrid 3.7 master. You can leave them running during the upgrade of the master to not interrupt your applications. Once the master upgrade is done, you should upgrade the IceGrid slaves to Ice 3.7 using the instructions above.

IceStorm Migration

Prerequisite: you need the IceStorm database export tool version 3.5 (`icestormdb35`) or version 3.6 (`icestormdb`). This utility is included in the Ice 3.6 distribution starting with Ice 3.6.2, but was not included in any Ice 3.5 distribution. If you are migrating from Ice 3.5, you need to build this export tool from sources: icestormdb35.

To start this migration, first stop the IceStorm server you wish to upgrade.

Then export the Freeze/Berkeley DB database environment of your IceStorm server:

With icestormdb 3.5 or 3.6

```
icestormdb --export icestormdb.ixp --dbhome /var/icestorm/db
```

The `icestormdb` export tool's version must match the existing IceStorm version; for example, use `icestormdb35` with an IceStorm 3.5.x.

The resulting file (`icestormdb.ixp` in our example) is a binary file with the full content of the IceStorm database.

> If you deployed IceStorm with IceGrid, the IceStorm database environment is typically specified through a Freeze `dbenv` descriptor, and the corresponding Berkeley DB home directory is in a subdirectory of your IceGrid node data directory.

The `icestormdb` utility can import this file into a Freeze/Berkeley DB or LMDB database.

Next, create a directory for your new IceStorm LMDB database files:

```
mkdir /var/icestorm/lmdb
```

Now import `icestormdb.ixp` into this new LMDB database environment directory:

With icestormdb 3.7 or greater

```
icestormdb --import icestormdb.ixp --dbpath /var/icestorm/lmdb
```

Finally, edit your IceStorm configuration file and replace the `Freeze.DbEnv.<Service Name>.DBHome` property with the new property `<Service Name>.LMDB.Path`.

> When IceStorm is deployed through IceGrid, a typical and recommended directory for this LMDB database is `${service.data}`.

> While the instructions above are sufficient for most deployments, you may want to review IceStorm Persistent Data and IceStorm Database Utility for detailed information about the tool and LMDB configuration options.

Changed APIs

This section describes APIs common to multiple language mappings (often specified using Slice) that have changed, potentially in ways that are incompatible with previous releases.

Forward Declared Slice Interfaces and Classes

A forward declared interface or class must be fully defined in the same Slice translation unit if any Slice definition in this translation unit uses a proxy of this interface (or class), or uses this forward declarated class in a non-local context (typically as a parameter in a non local operation).

Connection and Endpoint Information

The local classes `Ice::EndpointInfo` and `Ice::ConnectionInfo`, and all derived classes (`Ice::IPEndpointInfo`, `IceSSL::EndpointInfo`, etc.) were refactored. These classes now support an `underlying` data member that provides information on the underlying transport. For example, the `ssl` transport is based on the `tcp` transport so the `underlying` data member of an `ssl` endpoint or connection will contain an instance of `Ice::TCPEndpointInfo` or `Ice::TCPConnectionInfo`. See Using Connections for additional information.

Connection Changes

The API for Ice::Connection has changed in several ways:

- there are now separate callbacks for the close and heartbeat callbacks
- `Connection::close` now accepts an enum parameter instead of a bool

Flushing Batch Requests

These operations now take an additional parameter to control compression.

Classes no longer derive from Object

(Affects all language mappings except: C++98, Java Compat, Objective-C)

In Ice 3.6 and prior releases, a Slice class derives implicitly from `Object`, just like Slice interfaces. In Ice 3.7, Slice classes derive implicitly from `Value` (a new keyword). Slice interfaces still implicitly inherit from `Object`.

When mapped to C#, C++ with the C++11 mapping, Java, JavaScript, Python and more, the corresponding mapped native class no longer derives from `Ice::Object`. It derives instead from `Ice::Value`. Let's take an example:

Slice

```
module M
{
    class A
    {
        string x;
    };
};
```

This Slice class `A` is mapped to:

C# JavaScript Python

```
// 3.6
public partial class A : Ice.Object
{
    ...
}

// 3.7
public partial class A : Ice.Value
{
    ...
}
```

```
// 3.6
M.A = class extends Ice.Object
{
    ...
};

// 3.7
M.A = class extends Ice.Value
{
    ...
};
```

```
# 3.6
class A(Ice.Object):
    ...

# 3.7
class A(Ice.Value):
    ...
```

> In the language mappings unaffected by this change - C++98, Java Compat and Objective-C - `Value` and `Object` are mapped to the same native class.

Moreover, an operation on a Slice class is no longer mapped to an abstract method on the corresponding native class: the Slice compiler generates instead a separate skeleton class (typically with a `Disp` suffix) with the mapped method.

In a similar fashion, the mapped class for a Slice class that implements an interface no longer implements anything related to this interface. The Slice compiler generates instead a separate, independent skeleton class that implements the mapped interface.

Interface Operation Parameters

(Affects all language mappings except: C++98, Java Compat, Objective-C)

In Ice 3.6 and prior releases, you could use an interface as the type for an operation parameter, for example:

Slice

```
module M
{
    interface Marker
    {
        string print();
    };

    interface Receiver
    {
        Marker op(Marker x); // Marker not Marker*, i.e. pass-by-value
    };

    class A implements Marker
    {
        string msg;
    };
};
```

This way, only instances of Slice classes that implement this interface would be accepted as a parameter to this operation.

With Ice 3.7, the Slice definitions above remain valid, but the parameters are now mapped like `Value`s. For example, the `Receiver` interface is mapped to the following proxy and skeleton classes in C#:

C#

```
// Proxy
public interface ReceiverPrx : Ice.ObjectPrx
{
    Ice.Value op(Ice.Value x, Ice.OptionalContext context = new
Ice.OptionalContext());
    ...
}

// Skeleton
public abstract class ReceiverDisp_ : Ice.ObjectImpl, Receiver
{
    public abstract Ice.Value op(Ice.Value x, Ice.Current current =
null);
    ...
}
```

In the unusual situation where you need to send an interface "instance" by value, where the value carries only the interface's type id, each language mapping provides a new helper class for this purpose named `InterfaceByValue`.

Removed APIs

The following APIs were removed in this release:

- Interface `Ice::ConnectionCallback` (replaced by `Ice::HeartbeatCallback` and `Ice::CloseCallback`)
- Operation `Ice::Connection::setCallback` (replaced by the `setCloseCallback` and `setHeartbeatCallback` operations)
- Exception `Ice::NoObjectFactoryException` (replaced by `Ice::NoValueFactoryException`)
- Exception `Ice::ForcedCloseConnectionException` (replaced by `Ice::ConnectionManuallyClosedException`)
- Class `Ice::UnknownSlicedObject` (replaced by `Ice::UnknownSlicedValue`)

Deprecated APIs

This section describes the APIs that are deprecated in this Ice release, and will be removed in a future release. If your application uses one or more of the APIs listed below, we recommend updating it as soon as possible.

Operations on Classes

Operations on Slice classes are now deprecated: when feasible, you should convert these classes to interfaces (with only operations and no data members) or to classes without operations. If you need to keep classes with operations for interoperability with older applications, the global metadata directive `suppress-warning:deprecated` allows you to compile your Slice files without warnings.

Likewise, having a Slice class implement one or more interfaces is now deprecated.

> Local classes with operations are not deprecated. Such local classes are used by some Ice APIs, such as `Ice::EndpointInfo`.

Object Factories

Object factories are now referred as value factories following the deprecation of classes with operations. As a result, the following `Ice::Communicator` operations have been deprecated:

- `Communicator::addObjectFactory`
- `Communicator::findObjectFactory`

You should now use the `ValueFactoryManager` interface returned by `Communicator::getValueFactoryManager` to manage value factories.

Exception `ice_name` Method

The `ice_name` method for Ice exceptions has been deprecated in the various language mappings. It has been replaced by a new `ice_id` method.

Thread Hook in Python and C#

The `threadHook` member of `InitializationData` is deprecated. The new members `threadStart` and `threadStop` can be set to callable objects (Python) or `System.Action` delegates (C#).

Batch Request Interceptor in Python

The `Ice.BatchRequestInterceptor` class is deprecated. The `batchRequestInterceptor` member of `InitializationData` can be set to a callable object.

IcePatch2

IcePatch2 and IceGrid's distribution mechanism (based on IcePatch2) are now deprecated.

cpp:type:string and cpp:type:wstring Metadata on Modules

The metadata `cpp:string` and `cpp:wstring` are deprecated as module metadata, but not on other scopes.

Upgrading your Application from Ice 3.5

In addition to the information provided in Upgrading your Application from Ice 3.6, users who are upgrading from Ice 3.5 should also review this page.

On this page:

Timeout changes

In previous Ice versions, the timeout set for a connection also served as the timeout for all invocations on that connection, such that an invocation timeout would cause Ice to close the connection and consequently report a timeout exception for all pending invocations on that connection.

With the addition of invocation timeouts, Ice now provides a much cleaner separation between two distinct features:

- Connection timeouts
  These timeouts should be used as a fail-safe strategy for handling unrecoverable network errors in a timely fashion.

- Invocation timeouts
  You can now safely abort an invocation that's taking too long to complete without affecting other invocations pending on the same connection.

Together with the new heartbeat functionality offered by Ice's Active Connection Management facility, applications now have much more

control over their connections.

Existing Ice applications could configure connection timeouts in several ways, such as by setting Ice.Override.Timeout, or with the `-t` option in endpoints, or by calling `ice_timeout` on proxies. Ice still supports all of these options, and they all configure connection timeouts just like in previous Ice versions. However, you should carefully review these settings to see that they match the true purpose of connection timeouts. Specifically, connection timeouts should normally be chosen based on the speed of the network on which a connection takes place; a small timeout value is fine for a relatively fast network, but a larger value may be necessary for slower connections. Ice enforces connection timeouts when performing network operations such as connection establishment, reading, writing, and closing; setting a timeout allows Ice (and therefore the application) to detect a network problem within a well-defined time period without waiting for low-level network protocols to detect and report the issue.

We recommend that every application enable connection timeouts, and as a result they are now enabled by default as defined by the new property `Ice.Default.Timeout`.

The timeout semantics in previous Ice versions made it difficult for developers to employ invocation timeouts correctly. Using a small timeout value in order to detect network issues reasonably soon risked the possibility that some invocations might time out prematurely. These conflicting goals may have caused developers to modify their application designs to avoid long-running invocations. With the introduction of a separate invocation timeout feature, it's now possible to set connection timeouts appropriately for network errors while using invocation timeouts only for those operations that require them. Unlike connection timeouts, where setting different timeout values causes Ice to open separate connections, invocation timeouts of various values can be freely mixed on the same connection.

The default invocation timeout in Ice 3.6 is "infinite", meaning invocations do not time out by default. You can change the default setting using the new property `Ice.Default.InvocationTimeout`. For individual proxies, the method `ice_invocationTimeout` returns a new proxy configured with the desired invocation timeout. If you need to continue using the timeout semantics of previous Ice versions, set `Ice.Default.InvocationTimeout` to `-2` or call `ice_invocationTimeout` with the value `-2` on a given proxy.

Note that retry semantics differ between connection and invocation timeouts. Ice still performs automatic retries if possible for connection timeouts, but does not retry invocations that fail due to an invocation timeout.

ACM changes

The Active Connection Management (ACM) facility now offers additional control over its behavior, along with a new automatic heartbeat feature. Client-side ACM was enabled by default in prior Ice versions, which means an existing Ice application most likely uses ACM unless the application explicitly disabled it.

There are several ACM changes that may affect an existing application:

- The `Ice.ACM.Client` and `Ice.ACM.Server` properties have been deprecated and replaced by `Ice.ACM.Client.Timeout` and `Ice.ACM.Server.Timeout`, respectively.
- With the new heartbeat feature, Ice automatically sends heartbeat messages at regular intervals. You can use this feature in place of a dedicated background thread that was commonly used in previous versions of Ice for keeping a session or a bidirectional connection active.
- It's no longer necessary to disable ACM in Glacier2 clients as long as you enable ACM heartbeats.
- Server-side ACM is now enabled by default. Like with the previous versions, the default server-side configuration doesn't close idle connections. However, it now enables heartbeats while incoming requests are pending. This ensures the client doesn't close the connection prematurely while there are long invocations pending.

IceGrid and Glacier2 sessions

The ACM changes in Ice 3.6 offer new implementation strategies for applications that create Glacier2 or IceGrid sessions. Since a session is tightly bound to a connection, we strongly recommended in prior releases that you disable ACM altogether to avoid the risk of ACM prematurely closing a connection and consequently terminating a session. As of Ice 3.6, it's no longer necessary to disable ACM in these situations. Furthermore, the addition of ACM heartbeats means you can remove existing code that creates a background thread just to keep a session alive.

The `Ice::Connection` interface provides several new operations, including the ability to obtain and modify a connection's current ACM settings. If you use the Glacier2 helper classes, they call the `Connection` operations to tailor the ACM timeout and heartbeat based on the router's ACM configuration. Applications that manually create a Glacier2 session can configure ACM like this:

C++

```
Glacier2::RouterPrx router = ...
// Create a session...
int acmTimeout = router->getACMTimeout();
if(acmTimeout > 0)
{
    Ice::ConnectionPtr connection = router->ice_getCachedConnection();
    connection->setACM(acmTimeout, IceUtil::None, Ice::HeartbeatAlways);
}
```

The new operation `Glacier2::Router::getACMTimeout` returns the router's server-side ACM timeout. In this example, the client calls `setACM` on its connection to the router, passing this same timeout value to ensure consistency between client and server. The client also enables automatic heartbeats so that the connection remains active and prevents the router's server-side ACM from closing the connection.

The ACM improvements include changes to the ACM configuration properties. You can use these properties to achieve the same result as the code above, however the properties can potentially affect other connections as well.

Finally, another new operation on the `Connection` interface lets you specify a callback object that will be notified when the connection receives a heartbeat message, and when the connection closes. This feature can be especially useful for session-based applications that need to closely monitor their connections.

Glacier2 compatibility for helper classes

The Glacier2 helper classes in Ice 3.6 now depend on the ACM heartbeat features described above to keep a session alive. This functionality requires a Glacier2 router that also uses Ice 3.6 or later. If you're upgrading a client to Ice 3.6, we strongly recommend upgrading the Glacier2 router to 3.6 as well. Using an earlier version of Glacier2 will require your application to manually keep the session alive.

SSL changes

IceSSL for C++ has been overhauled to make use of platform-native SSL APIs where possible:

- IceSSL on Windows now uses SChannel
- IceSSL on OS X now uses Secure Transport
- Linux platforms continue to use OpenSSL as in previous releases

As a result, there have been a number of changes to the IceSSL configuration properties and its C++ API. In the IceSSL property reference, you'll see platform differences marked with **C++ using Windows**, **C++ using OS X**, and **C++ using OpenSSL**, respectively.

We discuss the affected platforms below, along with other general SSL changes.

> These changes also affect applications that use Ice for Python, Ice for Ruby, and Ice for PHP because these language mappings are based on Ice for C++.

SSLv3 disabled by default

To improve security, IceSSL now disables the SSLv3 protocol by default. In other words, only TLS protocols are enabled by default.

Although we do not recommend it, you can enable SSLv3 using the following settings:

```
# Enable only SSLv3
IceSSL.Protocols=SSL3

# Enable SSLv3 and TLS
IceSSL.Protocols=SSL3, TLS1_0, TLS1_1, TLS1_2

# OS X only: Enables SSLv3 and TLS
IceSSL.ProtocolVersionMin = "SSLv3"
```

Refer to IceSSL.* for more information on these settings.

Certificate verification

The default value of the `IceSSL.VerifyDepthMax` property is now three (it was previously two). This allows certificate chains of three certificates, such as a chain consisting of a peer certificate, a CA certificate, and a Root CA certificate.

The certificate chain provided in `IceSSL::ConnectionInfo` should now always include the root certificate if the chain is successfully verified.

We also added a `verified` member to `IceSSL::ConnectionInfo`, which indicates whether or not the peer certificate was successfully verified. For a client that sets `IceSSL.VerifyPeer=0`, this member allows the client to check the verification status of the server's certificate. For server connections, the member should always be true since servers always reject invalid client certificates.

SSL changes on OS X

By using Secure Transport, IceSSL on OS X has now become very similar to IceSSL in Ice Touch. For example, you can use OS X keychains, take advantage of the system's default graphical password prompt, and use many of the same configuration properties.

General Changes

- Password prompt
  In previous releases, OpenSSL would attempt to prompt command-line users for a password if the application failed to define a password or supply a password callback. Now OS X will use its default graphical password prompt in this situation.
- DSA certificates
  DSA certificates are no longer supported on OS X. If you used DSA certificates, you will need to generate RSA equivalents. The `IceSSL.CertFile` property still accepts the same syntax in that it allows you to specify two files (one RSA certificate file and one DSA certificate file), but it will ignore the DSA certificate.
- MD5 signatures
  OS X does not support certificates with MD5 signatures. We recommend using SHA256 instead.

API Changes

- Since IceSSL on OS X no longer uses OpenSSL, the native C++ class `IceSSL::Plugin` does not support the `setContext` and `getContext` methods.
- The method `IceSSL::Certificate::verify(const PublicKeyPtr&)` has been deprecated. The new method `IceSSL::Certificate::verify(const CertificatePtr&)` takes its place.

Property Changes

- Keychains
  The properties `IceSSL.Keychain` and `IceSSL.KeychainPassword` are now supported on OS X.
- Certificate authorities

Use the new property `IceSSL.CAs` to denote a file containing the certificates of your trusted certificate authorities. If you'd rather use the system's default certificate authorities, enable `IceSSL.UsePlatformCAs` instead.

- Certificates
  Use the `IceSSL.CertFile` property to denote a PKCS12 file containing both a certificate and a private key. The `IceSSL.KeyFile` property is now deprecated and should no longer be used to denote a separate private key file.
- Certificate queries
  The new property `IceSSL.FindCert` lets you query a keychain to find a certificate matching certain criteria.
- Diffie Hellman parameters
  The new property `IceSSL.DHParams` allows you to specify the name of a file containing pre-generated Diffie Hellman parameters. The property `IceSSL.DH.bits` is no longer supported on OS X.
- Protocol limits
  The `IceSSL.Protocols` property is no longer supported on OS X for specifying the versions of SSL/TLS that a program will accept. Use the new properties `IceSSL.ProtocolVersionMin` and `IceSSL.ProtocolVersionMax` instead.

Refer to deprecated APIs for information about obsolete properties.

SSL changes on Windows

By using SChannel, IceSSL on Windows has now become very similar to IceSSL for .NET. For example, you can use certificate stores and many of the same configuration properties.

General Changes

- Password prompt
  In previous releases, OpenSSL would attempt to prompt command-line users for a password if the application failed to define a password or supply a password callback. Windows does not have a default password prompt, so this situation will now result in a run-time exception.
- Anonymous Diffie Hellman (ADH)
  ADH ciphers are no longer supported on Windows.
- Certificate formats
  Windows is able to load a certificate in PEM format if no password is required. Use the PFX (PKCS#12) format for password-protected certificates and keys.

API Changes

- Since IceSSL on Windows no longer uses OpenSSL, the native C++ class `IceSSL::Plugin` does not support the `setContext` and `getContext` methods.
- The method `IceSSL::Certificate::verify(const PublicKeyPtr&)` has been deprecated. The new method `IceSSL::Certificate::verify(const CertificatePtr&)` takes its place.

Property Changes

- Certificate authorities
  Use the new property `IceSSL.CAs` to denote a file containing the certificates of your trusted certificate authorities. If you'd rather use the system's default certificate authorities, enable `IceSSL.UsePlatformCAs` instead.
- Certificates
  Use the `IceSSL.CertFile` property to denote a PKCS12 file containing both a certificate and a private key. The `IceSSL.KeyFile` property is now deprecated and should no longer be used to denote a separate private key file.
- Certificate queries
  The new property `IceSSL.FindCert` lets you query a certificate store to find a certificate matching certain criteria.

Refer to deprecated APIs for information about obsolete properties.

SSL changes on Linux

API Changes

- The method `IceSSL::Certificate::verify(const PublicKeyPtr&)` has been deprecated. The new method `IceSSL::Certificate::verify(const CertificatePtr&)` takes its place.

Property Changes

- Certificate authorities
  Use the new property `IceSSL.CAs` to denote a file containing the certificates of your trusted certificate authorities. If you'd rather use the system's default certificate authorities, enable `IceSSL.UsePlatformCAs` instead.
- Certificates
  Use the `IceSSL.CertFile` property to denote a PKCS12 file containing both a certificate and a private key. The `IceSSL.KeyFile` property is now deprecated and should no longer be used to denote a separate private key file.

Refer to deprecated APIs for information about obsolete properties.

Collocated Invocation changes

Ice has always supported location transparency, where a proxy invocation should ideally have the same semantics regardless of whether the target object is on a different host, a different process on the same host, or collocated in the current process. For this latter case, a collocated invocation is defined as a proxy invocation on a target object that is hosted by an object adapter in the same process and created using the same communicator as the proxy. In previous versions of Ice, the semantics of collocated invocations differed in several ways from regular "remote" invocations:

- Classes and exceptions were never sliced. Instead, the receiver always received a class or exception as the derived type that was sent by the sender.
- If a collocated invocation threw an exception that was not in an operation's exception specification, the original exception was raised in the client instead of `UnknownUserException`. (This applied to the C++ mapping only.)
- Class factories were ignored.
- Invocation timeouts were ignored.
- If an operation implementation used an in parameter that was passed by reference as a temporary variable, the change affected the value of the in parameter in the caller (instead of modifying a temporary copy of the parameter on the callee side only).
- Asynchronous semantics were not supported for collocated invocations.

Ice 3.6 eliminates all of these differences. Most notably, you can now use the asynchronous invocation API on a collocated servant, and collocated invocations are now supported in Python.

Note however that a few differences in semantics still remain:

- Most collocated invocations are dispatched in the server-side thread pool just like regular invocations; the only exceptions are synchronous twoway collocated invocations with no invocation timeout, which are dispatched in the calling thread.
- The state of the servant's object adapter is ignored: collocated invocations proceed normally even if the servant's adapter is not yet activated or is in the holding state.
- AMI callbacks for asynchronous collocated invocations are dispatched from the servant's thread and not from the client-side thread pool unless AMD is used for the servant dispatch. In this case, the AMI callback is called from the client-side thread pool.
- Invocation timeouts work as usual, but connection timeouts are ignored.

If your application relies on collocated invocations, test it carefully with Ice 3.6 to ensure that it still behaves as expected.

Batch Invocation changes

We made some significant changes to the way batch invocations work in Ice 3.6. The following table compares the behavior of batch invocations between previous releases and Ice 3.6:

Batch invocation behavior in Ice 3.5 and earlier	Batch invocation behavior in Ice 3.6

A batch request is queued by the connection associated with the proxy.	A batch request is queued by the proxy used for the invocation. The only exception is for a fixed proxy , in which case batch requests continue to be queued by the connection associated with the proxy.
If a proxy was not already associated with a connection, the initial batch request could trigger a connection attempt.	A batch request does not trigger any network activity.
Calling `Communicator::flushBatchRequests` flushes all queued requests for all connections.	Calling `Communicator::flushBatchRequests` only flushes queued requests invoked using fixed proxies.
Calling `Connection::flushBatchRequests` flushes queued requests for all proxies associated with that connection.	Calling `Connection::flushBatchRequests` only flushes queued requests invoked using fixed proxies associated with that connection.
A batch of requests is compressed when at least one of the proxies used to create the requests in this batch has compression enabled.	A batch of requests is compressed when the proxy used to flush this batch has compression enabled. Batched requests flushed through `Communicator::flushBatchRequests` or `Connection::flushBatchRequest` never use compression.
Batch requests are not affected by proxy lifecycles because the requests are queued by connections.	Batch requests queued by a proxy are discarded when the proxy is deallocated.
Batch requests could be silently lost if an error occurred during a flush.	Calling `ice_flushBatchRequests` on a regular (non-fixed) proxy behaves like a oneway request: failures that occur during network activity trigger automatic retries and can eventually raise an exception. Furthermore, the `sent` callback is invoked for asynchronous calls to `ice_flushBatchRequests`.

With these changes, Ice's batch invocation facility has become more reliable and behaves more consistently with other invocation modes.

To give you more control over batch requests, we've also added a new `BatchInterceptor` API. You can configure a communicator with a custom batch interceptor in order to implement your own auto-flush algorithms or to receive notification when an auto-flush fails.

Logger changes

We added a new operation `getPrefix` to the local interface `Logger`.

Slice

```
module Ice {
local interface Logger {
...
     string getPrefix();
};
};
```

If you implement your own logger, you will need to update your implementation with a new `getPrefix`. `getPrefix` returns the prefix associated with this logger.

Crypt Password changes

Both the Glacier2 router and IceGrid registry provide a simple file-based authentication mechanism using a "crypt password" file that contains a list of user name and password-hash pairs. In Ice 3.5 and earlier releases, the Glacier2 router and IceGrid registry use the archaic DES-based Unix Crypt algorithm to hash the provided password and verify if this hash matches the hash in the password-file.

Ice 3.6 no longer supports this hash format on Windows and OS X. As a result, you need to regenerate your crypt password files on these platforms when upgrading to Ice 3.6.

C++ changes

Garbage collection changes

We've made significant changes to the garbage collection facility for cyclic object graphs. If your application uses this facility, note that we've removed the following:

- `Ice.GC.Interval` property
- `Ice.Trace.GC` property
- `Ice::collectGarbage()` function

The new property `Ice.CollectObjects` determines whether garbage collection is enabled by default for Slice class instances that are unmarshaled by the Ice run time. Refer to the garbage collection discussion for more information on using this feature.

String converter changes

A number of changes have been made to the C++ string conversion API in this release:

- The `stringConverter` and `wstringConverter` members of `Ice::InitializationData` have been removed. Applications should use the process-wide string converter API instead.
- The classes and functions have moved from the `Ice` namespace to the `IceUtil` namespace.
- Overloaded versions of the `nativeToUTF8` and `UTF8ToNative` convenience functions that accepted a communicator argument have been removed.
- The arguments have changed for the `stringToWstring` and `wstringToString` convenience functions.

OS X with C++

C++ developers on OS X need to be aware of several changes:

- C++11 libraries
  With Ice 3.5, C++11 applications needed to link with a separate set of C++11-specific Ice libraries located in `<Ice installation directory>/lib/c++11`. The Ice 3.6 binary distribution includes a single set of libraries that also support C++11, so you'll need to modify your application's library path to use `<Ice installation directory>/lib` instead.

- Minimum required version
  Ice 3.6 supports OS X 10.9 and 10.10, therefore the C++ libraries in the Ice binary distribution are built with `macosx-min-version=10.9`. Consequently, these libraries require `libc++` and are not compatible with `libstdc++`.

Java changes

Java mapping changes

The default constructor generated for Slice structures, exceptions and classes behaves differently for Ice 3.6 than in previous releases. Specifically, the default constructor now initializes string data members to an empty string, enumerator data members to the first enumerator in the enumeration, and structure data members to a default-constructed value.

In previous releases, the default constructor initialized these data members to null. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The Java mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure or enumerator prior to an invocation. In Ice 3.6, passing a null value for a structure or enumerator causes Ice to marshal a default-constructed

structure or the first enumerator, respectively.

New Java features

We have added several features that you may wish to incorporate into your Java application:

- Interrupts
- Buffers
- Java 8 and lambdas

C# changes

C# mapping changes

The default constructor generated for Slice structures, exceptions and classes behaves differently for Ice 3.6 than in previous releases. Specifically, the default constructor now initializes string data members to an empty string and structure data members to a default-constructed value.

In previous releases, the default constructor did not explicitly initialize these data members and so they had the default C# values. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The C# mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure prior to an invocation. In Ice 3.6, passing a null value for a structure causes Ice to marshal a default-constructed structure.

C# serialization changes

One of the new features in Ice 3.6 is support for .NET serialization for all Slice types (except proxies). Existing applications may be affected by this change because the `Ice.Optional` type now implements `ISerializable` and the serialized format of an optional value is different than in Ice 3.5.

Python changes

Developers who are migrating existing Ice applications from Ice 3.5 should be aware of a change that affects the Python language mapping. The `Ice.Unset` value now has `False` semantics, making it more convenient to test whether an optional Slice data member has a value:

Python

```
# Only valid with Ice 3.6!
if obj.optionalMember:
    # optionalMember has a value
```

With Ice 3.5, the code above would not have the intended behavior because the test would be true even when `optionalMember` is set to `Ice.Unset`. The correct way to write this using Ice 3.5 is shown below:

Python

```
# Correct test with Ice 3.5
if obj.optionalMember is not Ice.Unset:
    # optionalMember has a value
```

This code will also have the correct behavior with Ice 3.6, but the new style is easier to read. Also note that the Ice 3.6 semantics mean you need to use caution for optional values that can legally be set to `None`:

Python

```
if obj.optionalMember: # Fails for None AND Ice.Unset!
    # optionalMember is not Ice.Unset or None
```

You can distinguish between `Ice.Unset` and `None` as follows:

Python

```
if obj.optionalMember is Ice.Unset:
    # optionalMember is unset
elif obj.optionalMember is None:
    # optionalMember is set to None
else:
    # optionalMember is set to a value other than None
```

We recommend that you review for correctness all uses of `Ice.Unset` and tests of optional data members.

PHP changes

No changes to the PHP mapping in Ice 3.6 affect compatibility for existing applications.

Ruby changes

No changes to the Ruby mapping in Ice 3.6 affect compatibility for existing applications.

JavaScript changes

JavaScript mapping changes

The default constructor generated for Slice structures, exceptions and classes behaves differently for Ice 3.6 than in previous releases. Specifically, the default constructor now initializes string data members to an empty string and structure data members to a default-constructed value.

In previous releases, the default constructor initialized these data members to null. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The JavaScript mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure or enumerator prior to an invocation. In Ice 3.6, passing a null value for a structure or enumerator causes Ice to marshal a default-constructed structure or the first enumerator, respectively.

JavaScript packaging changes

The NodeJS packaging has changed from the original Ice for JavaScript 0.1 release, meaning existing JavaScript applications will need to modify their `require` statements. All of the top-level Ice modules (Ice, Glacier2, etc.) are now accessible by including the `ice` package:

JavaScript

```
var Ice = require("ice").Ice;
var Glacier2 = require("ice").Glacier2;
// ...
```

Loading the generated code for your own Slice definitions looks similar. Suppose we have the following definitions in `Hello.ice`:

Slice

```
module Demo {
interface Hello {
    idempotent void sayHello(int delay);
    void shutdown();
};
};
```

To make the `Demo` module conveniently accessible in our code, we can write:

JavaScript

```
var Demo = require("Hello").Demo;
var proxy = Demo.HelloPrx.uncheckedCast(...);
```

Nothing has changed for browser-based JavaScript applications, where loading `Ice.js` adds the Ice definitions to the global window object, and other Ice modules (`Glacier2.js`, etc.) must be loaded individually.

Note that Ice 3.6 adds support for the WebSocket transport to the Ice core, and includes new implementations of the WebSocket transport in Java and C#. This means you no longer need to use Glacier2 as an intermediary if your JavaScript client needs to communicate with a Java or C# server.

Objective-C changes

The default `init` method and convenience constructor generated for Slice structures, exceptions and classes behave differently for Ice 3.6 than in previous releases. Specifically, these methods now initialize string data members to an empty string, enumerator data members to the first enumerator, and structure data members to a default-constructed value.

In previous releases, the default constructor zero-initialized these data members. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The Objective-C mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure prior to an invocation. In Ice 3.6, passing a null value for a structure causes Ice to marshal a default-constructed structure.

Ubuntu packages

Users of Ice 3.5 on Ubuntu had the choice of using Debian's packages or ZeroC's own experimental packages. We called them "experimental" because we expected we might eventually change the packaging structure, and in fact we have changed the structure for Ice

3.6.

Upgrading an existing installation of the ZeroC packages for Ice 3.5 on Ubuntu to Ice 3.6 is relatively straightforward. First add the Ice repository to the system and update the package list:

```
$ sudo apt-add-repository "deb http://zeroc.com/download/Ice/3.6/ubuntu14.04 stable main"
$ sudo apt-get update
```

To upgrade all of the run-time packages:

```
$ sudo apt-get install zeroc-ice-all-runtime
```

To upgrade all of the development packages:

```
$ sudo apt-get install libzeroc-ice-dev libzeroc-ice-java zeroc-ice-all-dev
```

Refer to our binary distribution page for details on the individual packages.

Migrating IceGrid databases from Ice 3.5

Ice 3.6 supports the migration of IceGrid databases from Ice 3.3, 3.4 and 3.5. To migrate from earlier Ice versions, you will first need to migrate the databases to the Ice 3.3 format. If you require assistance with such migration, please contact support@zeroc.com.

To migrate, first stop the IceGrid registry you wish to upgrade.

Next, copy the IceGrid database environment to a second location:

```
$ cp -r db recovered.db
```

Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.3, use `db_recover` from Berkeley DB 4.6. For Ice 3.4, use `db_recover` from Berkeley DB 4.8. For Ice 3.5, use `db_recover` from Berkeley DB 5.3. You can verify the version of your `db_recover` tool by running it with the `-V` option:

```
$ db_recover -V
```

Now run the utility on your copy of the database environment:

```
$ db_recover -h recovered.db
```

Change to the location where you will store the database environments for IceGrid 3.6:

```
$ cd <new-location>
```

Next, run the `upgradeicegrid36.py` utility located in the `config` directory of your Ice distribution (or in `/usr/share/Ice-3.6` if using an RPM installation). The first argument is the path to the old database environment. The second argument is the path to the new database environment.

In this example we'll create a new directory `db` in which to store the migrated database environment:

```
$ mkdir db
$ upgradeicegrid36.py <path-to-recovered.db> db
```

Upon completion, the `db` directory contains the migrated IceGrid databases.

By default, the migration utility assumes that the servers deployed with IceGrid also use Ice 3.6. If your servers still use an older Ice version, you need to specify the `--server-version` command-line option when running `upgradeicegrid36.py`:

```
$ upgradeicegrid36.py --server-version 3.5.1 <path-to-recovered.db> db
```

The migration utility will set the `server` descriptor attribute `ice-version` to the specified version and the IceGrid registry will generate configuration files compatible with the given version.

If you are upgrading the master IceGrid registry in a replicated environment and the slaves are still running, you should first restart the master registry in read-only mode using the `--readonly` option, for example:

```
$ icegridregistry --Ice.Config=config.master --readonly
```

Next, you can connect to the master registry with `icegridadmin` or the IceGrid administrative GUI from Ice 3.6 to ensure that the database is correct. If everything looks fine, you can shutdown and restart the master registry without the `--readonly` option.

IceGrid slaves from Ice 3.3, 3.4 or 3.5 won't interoperate with the IceGrid 3.6 master. You can leave them running during the upgrade of the master to not interrupt your applications. Once the master upgrade is done, you should upgrade the IceGrid slaves to Ice 3.6 using the instructions above.

Migrating IceStorm databases from Ice 3.5

No changes were made to the database schema for IceStorm in this release. Furthermore, Ice 3.5 and Ice 3.6 use the same version of Berkeley DB (Berkeley DB 5.3.x). You can use IceStorm databases created with Ice 3.5 with Ice 3.6 without any transformation.

Migrating Freeze databases from Ice 3.5

No changes were made that would affect the content of your Freeze databases. Furthermore, Ice 3.5 and Ice 3.6 use the same version of Berkeley DB (Berkeley DB 5.3.x). You can use Freeze databases created with Ice 3.5 with Ice 3.6 without any transformation.

Migrating Android applications from Ice 3.5

Our recommended development environment for Android applications is Android Studio. Refer to the *Using the Binary Distribution* page appropriate for your platform for instructions on configuring a project in Android Studio.

Changed APIs

This section describes APIs whose semantics have changed, potentially in ways that are incompatible with previous releases.

The following APIs were changed in Ice 3.6:

- String conversion in C++
  Several changes were made to the string conversion API in C++.
- IceSSL
  The native IceSSL APIs changed on some platforms.
- C++ garbage collection
  The `Ice::collectGarbage` function was removed and replaced by a new garbage collection facility.

Removed APIs

This section generally describes APIs that were deprecated in a previous release and have now been removed. Your application may no longer compile or operate successfully if it relies on one of these APIs.

The following APIs were removed in Ice 3.6:

- Deprecated API for asynchronous method invocations (AMI)
  This API, which uses proxy operations such as `sayHello_async`, was deprecated in Ice 3.4 and is no longer available. The new API should be used instead. Refer to the appropriate language mapping section for more information.
- `Stats` facility
  This functionality was deprecated in Ice 3.5 and is now provided by the Instrumentation facility and the Metrics facet.
- `Ice.GC.Interval`
  `Ice.Trace.GC`
  `Ice::collectGarbage()`
  Significant changes were made to the C++ garbage collection facility.
- `Ice.MonitorConnections`

This setting is no longer necessary.

- `IceSSL::Plugin::setContext()`
  `IceSSL::Plugin::getContext()`
  These C++ methods are no longer available on Windows or OS X.

- `Ice::InitializationData::stringConverter`
  `Ice::InitializationData::wstringConverter`
  These C++ data members are no longer available. Use `IceUtil::setProcessStringConverter` and `IceUtil::setProcessWstringConverter` instead.

- `Ice::Router::addProxy()`
  `IceGrid::Admin::writeMessage()`
  `IceStorm::Topic::subscribe()`
  These Slice operations were deprecated in previous Ice releases.

- `IceUtil.Version`
  This Java class was deprecated in previous Ice releases. Use `Ice.Util.stringVersion()` and `Ice.Util.intVersion()` instead.

- `Ice::Object::ice_getHash()`
  This C++ method was deprecated in Ice 3.5.

- `IcePatch2.ChunkSize`
  `IcePatch2.Directory`
  `IcePatch2.Remove`
  `IcePatch2.Thorough`
  These properties were deprecated in previous Ice releases and replaced by properties that use the prefix `IcePatch2Client`.

- `Glacier2.AddSSLContext`
  This property was deprecated in Ice 3.3.1 and replaced by `Glacier2.AddConnectionContext`.

The following components were removed in Ice 3.6:

- Qt SQL database plug-ins for IceGrid and IceStorm, which were deprecated in Ice 3.5.
  Freeze is now the only persistence mechanism for these services.

Deprecated APIs

This section discusses APIs and components that are now deprecated. These APIs will be removed in a future Ice release, therefore we encourage you to update your applications and eliminate the use of these APIs as soon as possible.

The following APIs were deprecated in Ice 3.6:

- `Ice.ACM.Client`
  `Ice.ACM.Server`
  Use `Ice.ACM.Client.Timeout` and `Ice.ACM.Server.Timeout` instead. See Active Connection Management for more information.

- `IceSSL.CertAuthFile`
  Use the new property `IceSSL.CAs` to specify the path name of a PEM file containing the Root Certificate Authorities.

- `IceSSL.CertAuthDir` (OpenSSL only)
  Use the new property `IceSSL.CAs` to specify the path name of a directory containing the Root Certificate Authorities.

- `IceSSL.KeyFile`
  Use `IceSSL.CertFile` to configure the IceSSL identity using a PKCS12 file.

- `IceSSL.ImportCert.*` (.NET only)
  This property caused the IceSSL plug-in to import a certificate into a Windows certificate store. Going forward, users will need to install certificates in a store using Windows-provided tools if necessary. However, it is now possible to use trusted CA certificates from a file without loading them into a store. See `IceSSL.CAs` for more information.

- `IceSSL.PersistKeySet` (.NET only)
  Only used by the now-deprecated `IceSSL.ImportCert` property.

- `IceSSL.KeySet` (.NET only)
  Use `IceSSL.CertStoreLocation` instead.
- `IceSSL::Certificate::verify` (C++ only)
  The method `verify(const PublicKeyPtr&)` has been deprecated. The new method `verify(const CertificatePtr&)` takes its place.
- `IceBox.InstanceName`
  `IceBox.ServiceManager.`*`AdapterProperty`*
  These properties are no longer necessary because their functionality is provided by the service manager administrative facet.
- `clr:collection`
  `Ice.CollectionBase`
  `Ice.DictionaryBase`
  The `clr:collection` metadata tag, along with the C# classes `Ice.CollectionBase` and `Ice.DictionaryBase`, were originally provided for backward compatibility with Ice versions prior to 3.3 and are now deprecated. Existing applications should migrate their code to use the standard C# collection types.
- `Ice::StringConverterPlugin` (C++ only)
  The `StringConvererPlugin` base class is deprecated. You should use a regular `Ice::Plugin` to install your string converter(s).
- `Ice::CollocatedOptimizationException`
  This exception is no longer used now that collocated invocations have become much more flexible.
- `Ice.BatchAutoFlush`
  This property controlled whether the Ice run time would automatically flush batch requests for a connection after enough requests had been queued to reach the limit established by `Ice.MessageSizeMax`. We're deprecating this property in favor of a new one, `Ice.BatchAutoFlushSize`, whose value Ice now uses as the limit for automatic flushing. If your application sets `Ice.BatchAutoFlush=0` to disable automatic flushing, you can achieve the same behavior by setting `Ice.BatchAutoFlushSize=0`.

Visual C++ compiler warnings

The Ice 3.5 header files downgrade the following Visual C++ warnings to level 4:

C++

```
#   pragma warning( 4 : 4250 ) // ... : inherits ... via dominance
#   pragma warning( 4 : 4251 ) // class ... needs to have dll-interface
to be used by clients of class ...
```

This downgrade affects any file that includes these Ice header files.

Ice 3.6 no longer disables or downgrades any warning in your C++ code. As a result, when upgrading to Ice 3.6, your Ice application may produce compiler warnings that were not reported before. To eliminate these warnings, you can modify your source code, add `pragmas` or disable these warnings in your Visual Studio projects.

Upgrading your Application from Ice 3.4

In addition to the information provided in Upgrading your Application from Ice 3.5, users who are upgrading from Ice 3.4 should also review this page.

On this page:

Slicing behavior in Ice 3.5

In version 1.0 of the Ice encoding, Slice classes and user exceptions are always marshaled in a format that allows a receiver to "slice" an unknown derived type to a known base type. An application that uses Ice 3.4 or earlier may depend on this behavior. For example, the slicing feature makes it possible for a sender to evolve independently of a receiver, adding more derived types that the receiver does not understand without causing the receiver to fail should it encounter one of these new types.

Version 1.1 of the Ice encoding introduces two formats for classes and exceptions. The compact format, which is used by default, does not support the slicing feature, meaning an application that relies on this feature may begin to receive `NoObjectFactoryException` or `UnknownUserException` errors after upgrading to Ice 3.5. There are several possible solutions:

- Continue using version 1.0 of the Ice encoding by setting `Ice.Default.EncodingVersion` to `1.0` in all senders and receivers.
- If you wish to take advantage of other features offered by encoding version 1.1, you can make the sliced format the default by setting `Ice.Default.SlicedFormat` to 1 in all senders and receivers.
- Annotate your interfaces or operations with metadata so that the sliced format is only used where necessary.

Slice language changes in Ice 3.5

The term `optional` is now a Slice keyword in Ice 3.5, which means existing Slice definitions that use this term as an identifier will generate an error. To continue using `optional` as an identifier, you can use the Slice syntax for escaping keywords:

Slice

```
struct Example
{
    bool \optional;
};
```

C++ changes in Ice 3.5

The `slice2cpp` translator no longer generates C++ comparison operators for all Slice structures. Comparison operators are only generated for structures that qualify as a legal dictionary key type. To generate a comparison operator for a Slice structure that doesn't qualify as such,

you can prefix the `["cpp:comparable"]` metadata to your Slice struct definition.

Java changes in Ice 3.5

The generated code and supporting classes for the various Ice services are now provided as separate JAR files:

JAR File	Description
`Freeze.jar`	Freeze classes
`Glacier2.jar`	Glacier2 generated code and helper classes
`Ice.jar`	The core Ice run time, including the IceSSL plugin
`IceBox.jar`	IceBox server, generated code, admin utility
`IceGrid.jar`	IceGrid generated code
`IcePatch2.jar`	IcePatch2 generated code
`IceStorm.jar`	IceStorm generated code

Also note that the manifest in `Freeze.jar` no longer contains a reference to `db.jar`, so you may need to update your class path to include `db.jar`.

The Eclipse Plug-in still includes only `Ice.jar` by default, but you can easily include additional JAR files by changing the Slice2Java properties for your project.

Python changes in Ice 3.5

Ice 3.5 adds support for Python 3. Developers who are migrating existing Ice applications from Python 2 to Python 3 should be aware of the following changes that affect the Python language mapping:

- One of the most significant changes in Python 3 is its unification of the unicode and string types into just one type, `str`. Python developers must convert any existing unicode literals and variables into their string equivalents. For Ice developers, a benefit of this unification is that all string values sent "over the wire" are now sent and received as unicode strings. With Python 2.x, the Ice run time behavior remains unchanged: string and unicode values are both accepted as inputs, but only string values are returned as outputs.
- The Python 3 mapping for the Slice type `sequence<byte>` now defaults to the new Python type `bytes`. A string value is no longer accepted as a legal value for `sequence<byte>` with Python 3.

PHP changes in Ice 3.5

For compatibility with newer PHP releases, the Ice extension no longer supports "call-time" reference arguments when calling `Ice_initialize` and `Ice_createProperties`. For example, existing applications may invoke `Ice_initialize` with an explicit reference argument as follows:

PHP

```
$communicator = Ice_initialize(&$argv);
```

The proper way to call this function using Ice 3.5 is shown below:

PHP

```
$communicator = Ice_initialize($argv);
```

Ice-specific arguments are now removed from the given argument vector.

IPv6 changes in Ice 3.5

IPv6 is now enabled by default on all platforms. You can disable the use of IPv6 in Ice by setting the `Ice.IPv6` property to `0`, which may be necessary in environments that lack IPv6 support. The new property `Ice.PreferIPv6Address` determines whether IPv4 or IPv6 addresses take precedence when resolving host names.

Migrating IceGrid databases from Ice 3.4

Ice 3.5 supports the migration of IceGrid databases from Ice 3.3 and from Ice 3.4. To migrate from earlier Ice versions, you will first need to migrate the databases to the Ice 3.3 format. If you require assistance with such migration, please contact support@zeroc.com.

To migrate, first stop the IceGrid registry you wish to upgrade.

Next, copy the IceGrid database environment to a second location:

```
$ cp -r db recovered.db
```

Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` exec utable that matches the Berkeley DB version of your existing Ice release. For Ice 3.3, use `db_recover` from Berkeley DB 4.6. For Ice 3.4, use `db_recover` from Berkeley DB 4.8. You can verify the version of your `db_recover` tool by running it with the `-V` option:

```
$ db_recover -V
```

Now run the utility on your copy of the database environment:

```
$ db_recover -h recovered.db
```

Change to the location where you will store the database environments for IceGrid 3.5:

```
$ cd <new-location>
```

Next, run the `upgradeicegrid35.py` utility located in the `config` directory of your Ice distribution (or in `/usr/share/Ice-3.5` if using an RPM installation). The first argument is the path to the old database environment. The second argument is the path to the new database environment.

In this example we'll create a new directory `db` in which to store the migrated database environment:

```
$ mkdir db
$ upgradeicegrid35.py <path-to-recovered.db> db
```

Upon completion, the `db` directory contains the migrated IceGrid databases.

By default, the migration utility assumes that the servers deployed with IceGrid also use Ice 3.5. If your servers still use an older Ice version, you need to specify the `--server-version` command-line option when running `upgradeicegrid35.py`:

```
$ upgradeicegrid.py --server-version 3.4.2 <path-to-recovered.db> db
```

The migration utility will set the `server` descriptor attribute `ice-version` to the specified version and the IceGrid registry will generate configuration files compatible with the given version.

If you are upgrading the master IceGrid registry in a replicated environment and the slaves are still running, you should first restart the master registry in read-only mode using the `--readonly` option, for example:

```
$ icegridregistry --Ice.Config=config.master --readonly
```

Next, you can connect to the master registry with `icegridadmin` or the IceGrid administrative GUI from Ice 3.5 to ensure that the database is correct. If everything looks fine, you can shutdown and restart the master registry without the `--readonly` option.

IceGrid slaves from Ice 3.3 or 3.4 won't interoperate with the IceGrid 3.5 master. You can leave them running during the upgrade of the master to not interrupt your applications. Once the master upgrade is done, you should upgrade the IceGrid slaves to Ice 3.5 using the instruction above.

Migrating IceStorm databases from Ice 3.4

No changes were made to the database schema for IceStorm in this release. However, you still need to update your databases as described below.

Migrating Freeze databases from Ice 3.4

No changes were made that would affect the content of your Freeze databases. However, we upgraded the version of Berkeley DB and the new 1.1 encoding may require that you re-create your indices or set some additional configuration if you use Freeze maps. If you only use Freeze evictors, you only need to upgrade your databases to the new Berkeley DB format. See Encoding Version 1.1.

Upgrading to the Berkeley DB 5.3 format

When upgrading to Ice 3.5, you must upgrade your database to the Berkeley DB 5.3 format. The only change that affects Freeze is the format of Berkeley DB's log file.

The instructions below assume that the database environment to be upgraded resides in a directory named `db` in the current working directory. For a more detailed discussion of database migration, please refer to the Berkeley DB Upgrade Process.

To migrate your database:

1. Shut down the old version of the application.
2. Make a backup copy of the database environment:

```
> cp -r db backup.db      (Unix)
> xcopy /E db backup.db   (Windows)
```

3. Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.4, use `db_recover` from Berkeley DB 4.8. You can verify the version of your `db_recover` tool by running it with the `-V` option:

```
> db_recover -V
```

4. Use the `db_recover` tool to run recovery on the database environment:

```
> db_recover -h db
```

5. Recompile and install the new version of the application.
6. Force a checkpoint using the `db_checkpoint` utility. Note that you must use the `db_checkpoint` utility from Berkeley DB 5.3 when performing this step.

```
> db_checkpoint -1 -h db
```

7. Restart the application.

Freeze maps and the 1.1 encoding

The majority of Freeze maps will work out of the box with the new 1.1 encoding and without additional configuration. However there are a few cases where you will be required to take special action or add some configuration:

- If your Freeze maps use keys with Slice enumerations having more than 127 elements, the encoding of those keys will be incompatible with the 1.1 encoding. You will need to set the `Freeze.DbEnv.<env-name>.EncodingVersion` property to `1.0` to ensure that you decode the keys with the 1.0 encoding and not with the default 1.1 encoding.
- If your Freeze maps use indices and those indices use Slice classes, or use Slice enumerations having more than 127 elements, you have two options:
  1. Re-create the indices with the 1.1 encoding. To recreate your indices, you have to recreate the Freeze maps using the `recreate` method generated on the Freeze map classes by `slice2freezej`. See the Freeze Maps documentation for more information.
  2. Set the `Freeze.DbEnv.<env-name>.EncodingVersion` property to `1.0` to ensure that Freeze still uses the 1.0 encoding for your indices.
- If you use indices with Java Freeze maps and the map value uses classes, you have to recreate the indices because of a bug in the encoding of the Ice 3.4 indices. This only affects Java Freeze maps; C++ Freeze maps are not affected by this issue.

> If you configure your Freeze database environment to use the 1.0 encoding, you will not be able to take advantage of the new features provided with the 1.1 encoding, such as optional values and the new formats for classes.

Migrating Android applications from Ice 3.4

Prior versions of the Ice plug-in for Eclipse (including the version released for Ice 3.5b) created a workspace variable named `ICE_HOME` that referred to the Ice installation directory configured in Eclipse's *Preferences* dialog. After adding the Slice2Java builder to a project, the builder updated the project's build path to include Ice JAR files relative to `ICE_HOME`. For example, `Ice.jar` was configured as the library reference `ICE_HOME/lib/Ice.jar`.

As of Ice 3.5.0, the Eclipse plug-in no longer uses the `ICE_HOME` variable. Instead, it now uses the workspace variable `ICE_JAR_HOME` to refer to the subdirectory containing `Ice.jar`, equivalent to the previous setting of `ICE_HOME/lib`. Consequently, a new project's build path now includes the library reference `ICE_JAR_HOME/Ice.jar`.

The plug-in does not attempt to modify the build path of an existing Ice for Android project to use `ICE_JAR_HOME`, nor does it remove `ICE_HOME` from the workspace. An existing project can continue to use the `ICE_HOME` variable in its build path, but moving or copying the project to a new workspace where `ICE_HOME` is no longer defined means the project's build path must be updated.

Changed APIs in Ice 3.5

This section describes APIs whose semantics have changed, potentially in ways that are incompatible with previous releases.

The following APIs were changed in Ice 3.5:

- The dynamic invocation and dispatch API now requires encapsulations for input parameters and output results. For example, when calling `ice_invoke`, the value for `inParams` must be an encapsulation of the input parameters:

C++

```
Ice::OutputStreamPtr out = Ice::createOutputStream();
out->startEncapsulation();
// marshal parameters
out->endEncapsulation();
Ice::ByteSeq inParams;
out->finished(inParams);
// invoke operation...
```

  Similarly, the byte sequence containing the results must also be an encapsulation:

C++

```
Ice::ByteSeq results;
// invoke operation...
Ice::InputStreamPtr in = Ice::wrapInputStream(results);
in->startEncapsulation();
// unmarshal results
in->endEncapsulation();
```

- The streaming API uses new methods to mark the beginning and end of objects and exceptions: `startObject`/`endObject` and `startException`/`endException`.

Removed APIs in Ice 3.5

This section describes APIs that were deprecated in a previous release and have now been removed. Your application may no longer compile successfully if it relies on one of these APIs.

The following APIs were removed in Ice 3.5:

- `Ice.Default.CollocationOptimization`
  Use `Ice.Default.CollocationOptimized` instead.

- *`proxy`*`.CollocationOptimization`
  Use *`proxy`*`.CollocationOptimized` instead.

- *`adapter`*`.RegisterProcess`
  This property caused the Ice run time to register a proxy with the locator registry that allowed the process to be shut down remotely. The new administrative facility has replaced this functionality.

- `Ice.ServerId`
  As with *`adapter`*`.RegisterProcess`, this property was used primarily for IceGrid integration and has been replaced by a similar mechanism in the administrative facility.

- `Ice.Trace.Location`
  This property has been replaced by `Ice.Trace.Locator`.

- `Glacier2.Admin` and `IcePatch2.Admin`
  These are the names of administrative object adapters in Glacier2 and IcePatch2, respectively. The functionality offered by these object adapters has been replaced by that of the administrative facility, therefore these adapters (and their associated configuration properties) are no longer necessary.

- `Ice.Util.generateUUID()`
  In Java use `java.util.UUID.randomUUID().toString()`. In C# use `System.Guid.NewGuid.ToString()`.

- `Object.ice_hash()`
  `ObjectPrx.ice_getHash()`
  `ObjectPrx.ice_toString()`
  Ice no longer defines hash methods for objects and proxies. To convert a proxy into a string, use the standard platform methods: `toString` in Java, `ToString` in C#.

Deprecated APIs in Ice 3.5

This section discusses APIs and components that are now deprecated. These APIs will be removed in a future Ice release, therefore we encourage you to update your applications and eliminate the use of these APIs as soon as possible.

The following APIs were deprecated in Ice 3.5:

- `Stats` facility
  This functionality is now provided by the Instrumentation facility and the Metrics facet.

The following components were deprecated in Ice 3.5:

- Qt SQL database plug-ins for IceGrid and IceStorm
  Freeze is now the only supported persistence mechanism for these services.

Upgrading your Application from Ice 3.3

In addition to the information provided in Upgrading your Application from Ice 3.4, users who are upgrading from Ice 3.3 should also review this page.

On this page:

Java language mapping changes in Ice 3.4

The Java2 language mapping, which was deprecated in Ice 3.3, is no longer supported. The Slice compiler and Ice API now use the Java5 language mapping exclusively, therefore upgrading to Ice 3.4 may require modifications to your application's source code. The subsections below discuss the language mapping features that are affected by this change and describe how to modify your application accordingly.

Metadata

The global metadata directives `java:java2` and `java:java5` are no longer supported and should be removed from your Slice files. The Slice compiler now emits a warning about these directives.

Support for the portable metadata syntax has also been removed. This syntax allowed Slice definitions to define custom type metadata that the Slice compiler would translate to match the desired target mapping. For example:

Slice

```
["java:type:{java.util.ArrayList}"] sequence<String> StringList;
```

The braces surrounding the custom type `java.util.ArrayList` directed the Slice compiler to use `java.util.ArrayList<String>` in the Java5 mapping and `java.util.ArrayList` in the Java2 mapping.

All uses of the portable metadata syntax must be changed to use the corresponding Java5 equivalent.

Dictionaries

Now that Slice dictionary types use the Java5 mapping, recompiling your Slice files and your application may cause the Java compiler to emit "unchecked" warnings. This occurs when your code attempts to assign an untyped collection class such as `java.util.Map` to a generic type such as `java.util.Map<String, String>`. Consider the following example:

Slice

```
dictionary<string, int> ValueMap;

interface Table
{
    void setValues(ValueMap m);
};
```

A Java2 application might have used these Slice definitions as shown below:

Java

```
java.util.Map values = new java.util.HashMap();
values.put(...);

TablePrx proxy = ...;
proxy.setValues(values); // Warning
```

The call to `setValues` is an example of an unchecked conversion. We recommend that you compile your application using the compiler option shown below:

```
javac -Xlint:unchecked ...
```

This option causes the compiler to generate descriptive warnings about occurrences of unchecked conversions to help you find and correct the offending code.

Request Contexts

The Slice type for request contexts, `Ice::Context`, is defined as follows:

Slice

```
module Ice
{
    dictionary<string, string> Context;
};
```

As a dictionary, the `Context` type is subject to the same issues regarding unchecked conversions described for #Dictionaries. For example, each proxy operation maps to two overloaded methods, one that omits the trailing `Context` parameter and one that includes it:

Java

```
interface TablePrx
{
    void setValues(java.util.Map<String, Integer> m);  // No context

    void setValues(java.util.Map<String, Integer> m,
                   java.util.Map<String, String> ctx);
}
```

If your proxy invocations make use of this parameter, you will need to change your code to use the generic type shown above in order to eliminate unchecked conversion warnings.

Enumerations

The Java2 language mapping for a Slice enumeration generated a class whose API differed in several ways from the standard Java5 enum type. Consider the following enumeration:

Slice

```
enum Color { red, green, blue };
```

The Java2 language mapping for `Color` is shown below:

Java

```
public final class Color
{
    // Integer constants
    public static final int _red = 0;
    public static final int _green = 1;
    public static final int _blue = 2;

    // Enumerators
    public static final Color red = ...;
    public static final Color green = ...;
    public static final Color blue = ...;

    // Helpers
    public static Color convert(int val);
    public static Color convert(String val);
    public int value();

    ...
}
```

The first step in migrating to the Java5 mapping for enumerations is to modify all `switch` statements that use an enumerator. Before Java added native support for enumerations, the `switch` statement could only use the integer value of the enumerator and therefore the Java2 mapping supplied integer constants for use in `case` statements. For example, here is a `switch` statement that uses the Java2 mapping:

Java

```
Color col = ...;
switch(col.value())
{
case Color._red:
    ...
    break;
case Color._green:
    ...
    break;
case Color._blue:
    ...
    break;
}
```

The Java5 mapping eliminates the integer constants because Java5 allows enumerators to be used in `case` statements. The resulting code becomes much easier to read and write:

Java

```
Color col = ...;
switch(col)
{
case red:
    ...
    break;
case green:
    ...
    break;
case blue:
    ...
    break;
}
```

The next step is to replace any uses of the `value` or `convert` methods with their Java5 equivalents. The base class for all Java5 enumerations (`java.lang.Enum`) supplies methods with similar functionality:

Java

```
static Color[] values()          // replaces convert(int)
static Color valueOf(String val) // replaces convert(String)
int ordinal()                    // replaces value()
```

For example, here is the Java5 code to convert an integer into its equivalent enumerator:

Java

```
Color r = Color.values()[0]; // red
```

Note however that the `convert(String)` method in the Java2 mapping returned null for an invalid argument, whereas the Java5 enum method `valueOf(String)` raises `IllegalArgumentException` instead.

Refer to the manual for more details on the mapping for enumerations.

Changes to the Java API for Freeze maps in Ice 3.4

The Java API for Freeze maps has been revised to used Java5 generic types and enhanced to provide additional functionality. This section describes these changes in detail and explains how to migrate your Freeze application to the API in Ice 3.4.

General changes to Freeze maps in Java

The Freeze API is now entirely type-safe, which means compiling your application against Ice 3.4 is likely to generate unchecked conversion warnings. The generated class for a Freeze map now implements the `java.util.SortedMap<K, V>` interface, where *K* is the key type and *V* is the value type. As a result, applications that relied on the untyped `SortedMap` API (where all keys and values were treated as instances of `java.lang.Object`) will encounter compiler warnings in Ice 3.4.

For example, an application might have iterated over the entries in a map as follows:

Java

```
// Old API
Object key = new Integer(5);
Object value = new Address(...);
myMap.put(key, value);
java.util.Iterator i = myMap.entrySet().iterator();
while (i.hasNext())
{
    java.util.Map.Entry e = (java.util.Map.Entry)i.next();
    Integer myKey = (Integer)e.getKey();
    Address myValue = (Address)e.getValue();
    ...
}
```

This code will continue to work, but the new API is both type-safe and self-documenting:

Java

```
// New API
int key = 5;
Address value = new Address(...);
myMap.put(key, value); // The key is autoboxed to Integer.
for (java.util.Map.Entry<Integer, Address> e : myMap.entrySet())
{
    Integer myKey = e.getKey();
    Address myValue = e.getValue();
    ...
}
```

Although migrating to the new API may require some effort, the benefits are worthwhile because your code will be easier to read and less prone to defects. You can also take advantage of the "autoboxing" features in Java5 that automatically convert values of primitive types (such as `int`) into their object equivalents (such as `Integer`).

Please refer to the Freeze Manual for complete details on the new API.

Enhancements to Freeze maps in Java

Java6 introduced the `java.util.NavigableMap` interface, which extends `java.util.SortedMap` to add some useful new methods. Although the Freeze map API cannot implement `java.util.NavigableMap` directly because Freeze must remain compatible with Java5, we have added the `Freeze.NavigableMap` interface to provide much of the same functionality. A generated Freeze map class implements `NavigableMap`, as do the sub map views returned by map methods such as `headMap`. The `NavigableMap` interface is described in the Freeze Manual, and you can also refer to the Java6 API documentation.

Backward compatibility for Freeze maps in Java

The Freeze Map API related to indices underwent some significant changes in order to improve type safety and avoid unchecked conversion warnings. These changes may cause compilation failures in a Freeze application.

In the previous API, index comparator objects were supplied to the Freeze map constructor in a map (in Java5 syntax, this comparators map would have the type `java.util.Map<String, java.util.Comparator>`) in which the index name was the key. As part of our efforts to improve type safety, we also wanted to use the fully-specified type for each index comparator (such as `java.util.Comparator<Integer>`). However, given that each index could potentially use a different key type, it is not possible to retain the previous API while remaining type-safe.

Consequently, the index comparators are now supplied as data members of a static nested class of the Freeze map named `IndexComparators`. If your application supplied custom comparators for indices, you will need to revise your code to use `IndexComparators` instead. For example:

Java

```
// Old API
java.util.Map indexComparators = new java.util.HashMap();
indexComparators.put("index", new MyComparator());
MyMap map = new MyMap(..., indexComparators);

// New API
MyMap.IndexComparators indexComparators = new MyMap.IndexComparators();
indexComparators.valueComparator = new MyComparator();
MyMap map = new MyMap(..., indexComparators);
```

We also encourage you to modify the definition of your comparator classes to use the Java5 syntax, as shown in the example below:

Java

```
// Old comparator
class IntComparator implements java.util.Comparator
{
    public int compare(Object o1, Object o2)
    {
        return ((Integer)o1).compareTo(o2);
    }
}

// New comparator
class IntComparator implements java.util.Comparator<Integer>
{
    public int compare(Integer i1, Integer i2)
    {
        return i1.compareTo(i2);
    }
}
```

The second API change that might cause compilation failures is the removal of the following methods:

Java

```
java.util.SortedMap headMapForIndex(String name, Object key);
java.util.SortedMap tailMapForIndex(String name, Object key);
java.util.SortedMap subMapForIndex(String name, Object from, Object to);
java.util.SortedMap mapForIndex(String name);
```

Again, this API cannot be retained in a type-safe fashion, therefore `slice2freezej` now generates equivalent (and type-safe) methods for each index in the Freeze map class.

Please refer to the Freeze Manual for complete details on the new API.

Finalizers in Freeze

In previous releases, Freeze for Java used finalizers to close objects such as maps and connections that the application neglected to close. Most of these finalizers have been removed in Ice 3.4, and the only remaining finalizers simply log warning messages to alert you to the fact that connections and iterators are not being closed explicitly. Note that, given the uncertain nature of Java finalizers, it is quite likely that the remaining finalizers will not be executed.

Freeze packaging changes in Ice 3.4

All Freeze-related classes are now stored in a separate JAR file named `Freeze.jar`. As a result, you may need to update your build scripts, deployment configuration, and run-time environment to include this additional JAR file.

PHP changes in Ice 3.4

The Ice extension for PHP has undergone many changes in this release. The subsections below describe these changes in detail. Refer to the PHP Mapping for more information about the language mapping.

Static translation in PHP

In prior releases, Slice files were deployed with the application and loaded at Web server startup by the Ice extension. Before each page request, the extension directed the PHP interpreter to parse the code that was generated from the Slice definitions.

In this release, Slice files must be translated using the new compiler `slice2php`. This change offers several advantages:

- Applications may have more opportunities to improve performance through the use of opcode caching.
- It is no longer necessary to restart the Web server when you make changes to your Slice definitions, which is especially useful during development.
- Errors in your Slice files can now be discovered in your development environment, rather than waiting until the Web server reports a failure and then reviewing the server log to determine the problem.
- The development process becomes simpler because you can easily examine the generated code if you have questions about the API or language mapping rules.
- PHP scripts can now use all of the Ice local exceptions. In prior releases, only a subset of the local exception types were available, and all others were mapped to `Ice_UnknownLocalException`. See the section Run-time exceptions in PHP below for more information.

All of the Slice files for Ice and Ice services are translated during an Ice build and available for inclusion in your application. At a minimum, you must include the file `Ice.php`:

PHP

```
require 'Ice.php';
```

`Ice.php` contains definitions for core Ice types and includes a minimal set of generated files. To use an Ice service such as IceStorm, include the appropriate generated file:

PHP

```
require 'Ice.php';
require 'IceStorm/IceStorm.php';
```

Deploying a PHP application

With the transition to static code generation, you no longer need to deploy Slice files with your application. Instead, you will need to deploy the PHP code generated from your Slice definitions, along with `Ice.php`, the generated code for the Ice core, and the generated code for any Ice services your application might use.

Using communicators in PHP

In prior releases, each PHP page request could access a single Ice communicator via the `$ICE` global variable. The configuration of this communicator was derived from the profile that the script loaded via the `Ice_loadProfile` function. The communicator was created on demand when `$ICE` was first used and destroyed automatically at the end of the page request.

In this release, a PHP script must create its own communicator using an API that is similar to other Ice language mappings:

PHP

```
function Ice_initialize()
function Ice_initialize($args)
function Ice_initialize($initData)
function Ice_initialize($args, $initData)
```

`Ice_initialize` creates a new communicator using the configuration provided in the optional arguments. `$args` is an array of strings representing command-line options, and `$initData` is an instance of `Ice_InitializationData`.

An application that requires no configuration can initialize a communicator as follows:

PHP

```
$communicator = Ice_initialize();
```

More elaborate configuration scenarios are described in the section #PHP configuration below.

A script may optionally destroy its communicator:

PHP

```
$communicator->destroy();
```

At the completion of a page request, Ice by default automatically destroys any communicator that was not explicitly destroyed.

Using registered communicators in PHP

PHP applications may benefit from the ability to use a communicator instance in multiple page requests. Reusing a communicator allows the application to minimize the overhead associated with the communicator lifecycle, including such activities as opening and closing connections to Ice servers.

This release includes new APIs for registering a communicator in order to prevent Ice from destroying it automatically at the completion of a page request. For example, a session-based application can create a communicator, establish a Glacier2 session, and register the communicator. In subsequent page requests, the PHP session can retrieve its communicator instance and continue using the Glacier2 session.

The manual provides more information on this feature, and a new sample program can be found in `Glacier2/hello`.

PHP configuration

Prior releases supported four INI settings in PHP's configuration file:

- `ice.config`
- `ice.options`
- `ice.profiles`
- `ice.slice`

The `ice.slice` directive is no longer supported since Slice definitions are now compiled statically. The remaining options are still supported but their semantics are slightly different. They no longer represent the configuration of a communicator; instead, they define property sets that a script can retrieve and use to initialize a communicator.

The global INI directives `ice.config` and `ice.options` configure the default property set. The `ice.profiles` directive can optionally nominate a separate file that defines any number of named profiles, each of which configures a property set.

As before, the profiles use an INI file syntax:

```
[Name1]
config=file1
options="--Ice.Trace.Network=2 ..."

[Name2]
config=file2
options="--Ice.Trace.Locator=1 ..."
```

A new directive, `ice.hide_profiles`, overwrites the value of the `ice.profiles` directive as a security measure. This directive has a default value of 1, meaning it is enabled by default.

A script can obtain a property set using the new function `Ice_getProperties`. Called without an argument (or with an empty string), the function returns the default property set:

PHP

```
$props = Ice_getProperties();
```

Alternatively, you can pass the name of the desired profile:

PHP

```
$props = Ice_getProperties("Name1");
```

The returned object is an instance of `Ice_Properties`, which supports the standard Ice API.

For users migrating from an earlier release, you can replace a call to `Ice_loadProfile` as follows:

PHP

```
// PHP - Old API
Ice_loadProfile('Name1');

// PHP - New API
$initData = new Ice_InitializationData;
$initData->properties = Ice_getProperties('Name1');
$ICE = Ice_initialize($initData);
```

(Note that it is not necessary to use the symbol `$ICE` for your communicator. However, using this symbol may ease your migration to this release.)

`Ice_loadProfile` also installed the PHP definitions corresponding to your Slice types. In this release you will need to add `require` state ments to include your generated code.

Finally, if you wish to manually configure a communicator, you can create a property set using `Ice_createProperties`:

PHP

```
function Ice_createProperties($args=null, $defaultProperties=null)
```

`$args` is an array of strings representing command-line options, and `$defaultProperties` is an instance of `Ice_Properties` that supplies default values for properties.

As an example, an application can configure a communicator as shown below:

PHP

```
$initData = new Ice_InitializationData;
$initData->properties = Ice_createProperties();
$initData->properties->setProperty("Ice.Trace.Network", "1");
...
$ICE = Ice_initialize($initData);
```

PHP namespaces

This release includes optional support for PHP namespaces, which was introduced in PHP 5.3. Support for PHP namespaces is disabled by default; to enable it, you must build the Ice extension from source code with `USE_NAMESPACES=yes` (see `Make.rules` or `Make.rules.mak` in the `php/config` subdirectory). Note that the extension only supports one mapping style at a time; installing a namespace-enabled version of the extension requires all Ice applications on the target Web server to use namespaces.

With namespace support enabled, you must modify your script to include a different version of the core Ice types:

PHP

```
require 'Ice_ns.php'; // Namespace version of Ice.php
```

You must also recompile your Slice files using the `-n` option to generate namespace-compatible code:

```
% slice2php -n MySliceFile.ice
```

This mapping translates Slice modules into PHP namespaces instead of using the "flattened" (underscore) naming scheme. For example, `Ice_Properties` becomes `\Ice\Properties` in the namespace mapping. However, applications can still refer to global Ice functions by their traditional names (such as `Ice_initialize`) or by their namespace equivalents (`\Ice\initialize`).

Run-time exceptions in PHP

As mentioned earlier, prior releases of Ice for PHP only supported a limited subset of the standard run-time exceptions. An occurrence of an unsupported local exception was mapped to `Ice_UnknownLocalException`.

This release adds support for all local exceptions, which allows an application to more easily react to certain types of errors:

PHP

```
try
{
    $proxy->sayHello();
}
catch(Ice_ConnectionLostException $ex)
{
    // Handle connection loss
}
catch(Ice_LocalException $ex)
{
    // Handle other errors
}
```

This change represents a potential backward compatibility issue: applications that previously caught `Ice_UnknownLocalException` may need to be modified to catch the intended exception instead.

Downcasting in PHP

In prior releases, to downcast a proxy you had to invoke the `ice_checkedCast` or `ice_uncheckedCast` method on a proxy and supply a type ID:

PHP

```
$hello = $proxy->ice_checkedCast("::Demo::Hello");
```

This API is susceptible to run-time errors because no validation is performed on the type ID string. For example, renaming the `Hello` interface to `Greeting` requires that you not only change all occurrences of `Demo_Hello` to `Demo_Greeting`, but also fix any type ID strings that your code might have embedded. The PHP interpreter does not provide any assistance if you forget to make this change, and you will only discover it when that particular line of code is executed and fails.

To improve this situation, a minimal class is now generated for each proxy type. The purpose of this class is to supply `checkedCast` and un`checkedCast` static methods:

PHP

```
class Demo_HelloPrx
{
    public static function checkedCast($proxy, $facetOrCtx=null,
$ctx=null);

    public static function uncheckedCast($proxy, $facet=null);
}
```

Now your application can downcast a proxy as follows:

PHP
```
$hello = Demo_HelloPrx::checkedCast($proxy);
```

You can continue to use `ice_checkedCast` and `ice_uncheckedCast` but we recommend migrating your application to the new methods.

Other API changes for PHP

This section describes additional changes to the Ice API in this release:

- The global variable `$ICE` is no longer defined. An application must now initialize its own communicator as described above.
- Removed the following communicator methods:

  PHP
  ```
  $ICE->setProperty()
  $ICE->getProperty()
  ```

  The equivalent methods are:

  PHP
  ```
  $communicator->getProperties()->setProperty()
  $communicator->getProperties()->getProperty()
  ```

- Removed the following global functions:

  PHP
  ```
  Ice_stringToIdentity()
  Ice_identityToString()
  ```

  The equivalent methods are:

  PHP
  ```
  $communicator->stringToIdentity()
  $communicator->identityToString()
  ```

- These functions have also been removed:

  PHP
  ```
  Ice_loadProfile()
  Ice_loadProfileWithArgs()
  Ice_dumpProfile()
  ```

Refer to PHP configuration for more information.

Thread pool changes in Ice 3.4

A thread pool supports the ability to automatically grow and shrink as the demand for threads changes, within the limits set by the thread pool's configuration. In prior releases, the rate at which a thread pool shrinks was not configurable, but Ice 3.4.0 introduces the `ThreadIdleTime` property to allow you to specify how long a thread pool thread must remain idle before it terminates to conserve resources.

IceSSL changes in Ice 3.4

With the addition of the `ConnectionInfo` classes in this release, the `IceSSL::ConnectionInfo` structure has changed from a native type to a Slice class. This change has several implications for existing applications:

- As a Slice class, `IceSSL::ConnectionInfo` cannot provide the X509 certificate chain in its native form, therefore the chain is provided as a sequence of strings representing the encoded form of each certificate. You can use language-specific facilities to convert these strings back to certificate objects.
- For your convenience, we have added a native subclass of `IceSSL::ConnectionInfo` called `IceSSL::NativeConnectionInfo`. This class provides the certificate chain as certificate objects.
- The `CertificateVerifier` interface now uses `NativeConnectionInfo` instead of `ConnectionInfo`. If your application configures a custom certificate verifier, you will need to modify your implementation accordingly.
- In C++, also note that `NativeConnectionInfo`is now managed by a smart pointer, therefore the signature of the certificate verifier method becomes the following:

  C++
  ```
  virtual bool verify(const IceSSL::NativeConnectionInfoPtr&) = 0;
  ```

- The `getConnectionInfo` helper function has been removed because its functionality has been replaced by the `Connection::getInfo`operation. For example, in prior releases a C++ application would do the following:

  C++
  ```
  Ice::ConnectionPtr con = ...
  IceSSL::ConnectionInfo info = IceSSL::getConnectionInfo(con);
  ```

  Now the application should do this:

  C++
  ```
  Ice::ConnectionPtr con = ...
  IceSSL::ConnectionInfoPtr info =
  IceSSL::ConnectionInfoPtr::dynamicCast(con->getInfo());
  ```

  Alternatively, the application can downcast to the native class:

C++

```
Ice::ConnectionPtr con = ...
IceSSL::NativeConnectionInfoPtr info =
    IceSSL::NativeConnectionInfoPtr::dynamicCast(con->getInfo());
```

Migrating IceStorm and IceGrid databases from Ice 3.3

No changes were made to the database schema for IceStorm or IceGrid in this release. However, you still need to update your databases as described below.

Migrating Freeze databases from Ice 3.3

No changes were made that would affect the content of your Freeze databases. However, we upgraded the version of Berkeley DB, therefore when upgrading to Ice 3.4 you must also upgrade your database to the Berkeley DB 4.8 format. The only change that affects Freeze is the format of Berkeley DB's log file.

The instructions below assume that the database environment to be upgraded resides in a directory named `db` in the current working directory. For a more detailed discussion of database migration, please refer to the Berkeley DB Upgrade Process.

To migrate your database:

1. Shut down the old version of the application.
2. Make a backup copy of the database environment:

```
> cp -r db backup.db      (Unix)
> xcopy /E db backup.db   (Windows)
```

3. Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.3, use `db_recover` from Berkeley DB 4.6. You can verify the version of your `db_recover` tool by running it with the `-V`option:

```
> db_recover -V
```

4. Use the db_recover tool to run recovery on the database environment:

```
> db_recover -h db
```

5. Recompile and install the new version of the application.
6. Force a checkpoint using the `db_checkpoint` utility. Note that you must use the `db_checkpoint`utility from Berkeley DB 4.8 when performing this step.

```
> db_checkpoint -1 -h db
```

7. Restart the application.

Removed APIs in Ice 3.4.0

This section describes APIs that were deprecated in a previous release and have now been removed. Your application may no longer compile successfully if it relies on one of these APIs.

The following APIs were removed in Ice 3.4.0:

- `Glacier2.AddUserToAllowCategories`
  Use `Glacier2.Filter.Category.AcceptUser` instead.
- `Glacier2.AllowCategories`
  Use `Glacier2.Filter.Category.Accept` instead.
- `Ice.UseEventLog`
  Ice services (applications that use the C++ class `Ice::Service`) always use the Windows event log by default.
- `Communicator::setDefaultContext`
- `Communicator::getDefaultContext`
- `ObjectPrx:ice_defaultContext`
  Use the communicator's implicit request context instead.
- `nonmutating` keyword
  This keyword is no longer supported.
- `Freeze.UseNonmutating`
  Support for this property was removed along with the `nonmutating` keyword.
- `Ice::NegativeSizeException`
  The run time now throws `UnmarshalOutOfBoundsException` or `MarshalException` instead.
- `slice2docbook`
  This utility is no longer included in Ice.
- `Ice::AMD_Array_Object_ice_invoke`
  A new overloading of `ice_response` in the `AMD_Object_ice_invoke` class makes `AMD_Array_Object_ice_invoke` obsolete.
- Java2 mapping
  The Java2 mapping is no longer supported. Refer to Java language mapping changes in Ice 3.4 for more information.

Deprecated APIs in Ice 3.4.0

This section discusses APIs and components that are now deprecated. These APIs will be removed in a future Ice release, therefore we encourage you to update your applications and eliminate the use of these APIs as soon as possible.

The following APIs were deprecated in Ice 3.4.0:

- Asynchronous Method Invocation (AMI) interface
  The AMI interface in Ice 3.3 and earlier is now deprecated for C++, Java, and C#.
- `Glacier2.AddSSLContext`
  Replaced by `Glacier2.AddConnectionContext`.
- Standard platform methods should be used instead of the following:

  Java

  ```
  Ice.Object.ice_hash()         // Use hashCode
  Ice.ObjectPrx.ice_getHash()   // Use hashCode
  Ice.ObjectPrx.ice_toString()  // Use toString
  ```

  In Java, use `hashCode` and `toString`. In C#, use `GetHashCode` and `ToString`. In Ruby, use `hash` instead of `ice_getHash`.
- `Ice.Util.generateUUID()`

In Java use `java.util.UUID.randomUUID().toString()`. In C# use `System.Guid.NewGuid.ToString()`.

Upgrading your Application from Ice 3.2 or Earlier Releases

In addition to the information provided in Upgrading your Application from Ice 3.3, users who are upgrading from Ice 3.2 or earlier should also review this page.

On this page:

Migrating IceStorm databases from Ice 3.2

Ice 3.4 supports the migration of IceStorm databases from Ice 3.1 and from Ice 3.2. Migration from earlier Ice versions may work, but is not officially supported. If you require assistance with such migration, please contact support@zeroc.com.

To migrate, first stop your IceStorm servers.

Next, copy the IceStorm database environment to a second location:

```
$ cp -r db recovered.db
```

Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.1, use `db_recover` from Berkeley DB 4.3.29. For Ice 3.2, use `db_recover` from Berkeley DB 4.5. You can verify the version of your `db_recover` tool by running it with the `-V` option:

```
$ db_recover -V
```

Now run the utility on your copy of the database environment:

```
$ db_recover -h recovered.db
```

Change to the location where you will store the database environments for IceStorm 3.4:

```
$ cd <new-location>
```

Next, run the `icestormmigrate` utility. The first argument is the path to the old database environment. The second argument is the path to

the new database environment.

In this example we'll create a new directory `db` in which to store the migrated database environment:

```
$ mkdir db
$ icestormmigrate <path-to-recovered.db> db
```

Upon completion, the `db` directory contains the migrated IceStorm databases.

Migrating IceGrid databases from Ice 3.2

Ice 3.4 supports the migration of IceGrid databases from Ice 3.1 and from Ice 3.2. Migration from earlier Ice versions may work, but is not officially supported. If you require assistance with such migration, please contact support@zeroc.com.

To migrate, first stop the IceGrid registry you wish to upgrade.

Next, copy the IceGrid database environment to a second location:

```
$ cp -r db recovered.db
```

Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.1, use `db_recover` from Berkeley DB 4.3.29. For Ice 3.2, use `db_recover` from Berkeley DB 4.5. You can verify the version of your `db_recover` tool by running it with the `-V` option:

```
$ db_recover -V
```

Now run the utility on your copy of the database environment:

```
$ db_recover -h recovered.db
```

Change to the location where you will store the database environments for IceGrid 3.4:

```
$ cd <new-location>
```

Next, run the `upgradeicegrid.py` utility located in the `config` directory of your Ice distribution (or in `/usr/share/Ice-3.4.1` if using an RPM installation). The first argument is the path to the old database environment. The second argument is the path to the new database environment.

In this example we'll create a new directory `db` in which to store the migrated database environment:

```
$ mkdir db
$ upgradeicegrid.py <path-to-recovered.db> db
```

Upon completion, the `db` directory contains the migrated IceGrid databases.

By default, the migration utility assumes that the servers deployed with IceGrid also use Ice 3.4. If your servers still use an older Ice version, you need to specify the `--server-version` command-line option when running `upgradeicegrid.py`:

```
$ upgradeicegrid.py --server-version 3.2.1 <path-to-recovered.db> db
```

The migration utility will set the `server` descriptor attribute `ice-version` to the specified version and the IceGrid registry will generate configuration files compatible with the given version.

If you are upgrading the master IceGrid registry in a replicated environment and the slaves are still running, you should first restart the master registry in read-only mode using the `--readonly` option, for example:

```
$ icegridregistry --Ice.Config=config.master --readonly
```

Next, you can connect to the master registry with `icegridadmin` or the IceGrid administrative GUI to ensure that the database is correct. If everything looks fine, you can shutdown and restart the master registry without the `--readonly` option.

Migrating Freeze databases from Ice 3.2

No changes were made that would affect the content of your Freeze databases. However, we upgraded the version of Berkeley DB, therefore when upgrading to Ice 3.4 you must also upgrade your database to the Berkeley DB 4.8 format. The only change that affects Freeze is the format of Berkeley DB's log file.

The instructions below assume that the database environment to be upgraded resides in a directory named `db` in the current working directory. For a more detailed discussion of database migration, please refer to the Berkeley DB Upgrade Process.

To migrate your database:

1. Shut down the old version of the application.
2. Make a backup copy of the database environment:

   ```
   > cp -r db backup.db      (Unix)
   > xcopy /E db backup.db   (Windows)
   ```

3. Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.1, use `db_recover` from Berkeley DB 4.3.29. For Ice 3.2, use `db_recover` from Berkeley DB 4.5. You can verify the version of your `db_recover` tool by running it with the `-V`option:

   ```
   > db_recover -V
   ```

4. Use the `db_recover`tool to run recovery on the database environment:

   ```
   > db_recover -h db
   ```

5. Recompile and install the new version of the application.
6. Force a checkpoint using the `db_checkpoint` utility. Note that you must use the `db_checkpoint`utility from Berkeley DB 4.8 when performing this step.

   ```
   > db_checkpoint -1 -h db
   ```

7. Restart the application.

Removed APIs in Ice 3.3

This section describes APIs that were deprecated in a previous release and have been removed in Ice 3.3. Your application may no longer compile successfully if it relies on one of these APIs.

Please refer to Removed APIs in Ice 3.4.0 for information on APIs that were removed in Ice 3.4.

Thread per connection

The primary purpose of this concurrency model was to serialize the requests received over a connection, either because the application needed to ensure that requests are dispatched in the order they are received, or because the application did not want to implement the synchronization that might be required when using the thread pool concurrency model.

Another reason for using the thread-per-connection concurrency model is that it was required by the IceSSL plug-ins for Java and C#. This requirement has been eliminated.

The ability to serialize requests is now provided by the thread pool and enabled via a new configuration property:

`<threadpool>.Serialize=1`

Please refer to the manual for more details on this feature.

Aside from the potential semantic changes involved in migrating your application to the thread pool concurrency model, other artifacts of thread-per-connection may be present in your application and must be removed:

- The configuration properties `Ice.ThreadPerConnection` and `<proxy>.ThreadPerConnection`
- The proxy methods `ice_threadPerConnection` and `ice_isThreadPerConnection`

.NET metadata

The metadata directive `cs:collection` is no longer valid. Use `["clr:collection"]` instead.

C++

The following C++ methods have been removed:

- `Application::main(int, char*[], const char*, const Ice::LoggerPtr&)`

  Use `Application::main(int, char*[], const InitializationData&)` instead.

- `initializeWithLogger`
- `initializeWithProperties`
- `initializeWithPropertiesAndLogger`

  Use `initialize(int, char*[], const InitializationData&)` instead.

- `stringToIdentity`
- `identityToString`

  Use the equivalent operations on `Communicator`.

Java

The following methods have been removed:

- `Application.main(String, String[], String, Logger)`

  Use `Application.main(String, String[], InitializationData)` instead.

- `initializeWithLogger`
- `initializeWithProperties`
- `initializeWithPropertiesAndLogger`

Use `initialize(String[], InitializationData)` instead.

.NET

The following methods have been removed:

- `Application.main(string, string[], string, Logger)`

  Use `Application.main(string, string[], InitializationData)` instead.

- `initializeWithLogger`
- `initializeWithProperties`
- `initializeWithPropertiesAndLogger`

  Use `initialize(ref string[], InitializationData)` instead.

Python

The following methods have been removed:

- `initializeWithLogger`
- `initializeWithProperties`
- `initializeWithPropertiesAndLogger`

  Use `initialize(args, initializationData)` instead.

- `stringToIdentity`
- `identityToString`
  Use the equivalent operations on `Communicator`.

General

The following methods have been removed:

- `ice_hash`
- `ice_communicator`
- `ice_collocationOptimization`
- `ice_connection`
  These proxy methods were replaced by ones of the form `ice_get...`, such as `ice_getCommunicator`. `ice_collocationOptimization` is now `ice_getCollocationOptimized`.

- `ice_newIdentity`
- `ice_newContext`
- `ice_newFacet`
- `ice_newAdapterId`
- `ice_newEndpoints`
  These proxy methods were replaced by ones that do not use `new` in their names. For example, `ice_newIdentity` was replaced by `ice_identity`.

`Ice.LoggerPlugin`

This property provided a way to install a custom logger implementation. It has been replaced by a more generalized facility for installing custom loggers.

Deprecated APIs in Ice 3.3

This section discusses APIs and components that are deprecated in Ice 3.3. These APIs will be removed in a future Ice release, therefore we encourage you to update your applications and eliminate the use of these APIs as soon as possible.

Please refer to Deprecated APIs in Ice 3.4.0 for information on APIs that were deprecated in Ice 3.4.

Sequences as dictionary keys

The use of sequences, and structures containing sequences, as the key type of a Slice dictionary is now deprecated.

`LocalObject`

The mappings for the `LocalObject` type have changed in Java, .NET, and Python. The new mappings are shown below:

Java	`java.lang.Object`
.NET	`System.Object`
Python	`object`

The types `Ice.LocalObject` and `Ice.LocalObjectImpl` are deprecated.

`Ice.Trace.Location`

This property has been replaced by `Ice.Trace.Locator`.

`Ice.Default.CollocationOptimization`

This property, as well as the corresponding proxy property, have been replaced by `Ice.Default.CollocationOptimized` and `<proxy>.CollocationOptimized`, respectively.

`<Adapter>.RegisterProcess`

This property caused the Ice run time to register a proxy with the locator registry (e.g., IceGrid) that allowed the process to be shut down remotely. The new administrative facility has replaced this functionality.

`Ice.ServerId`

As with `<Adapter>.RegisterProcess`, this property was used primarily for IceGrid integration and has been replaced by a similar mechanism in the administrative facility.

`Glacier2.Admin` and `IcePatch2.Admin`

These are the names of administrative object adapters in Glacier2 and IcePatch2, respectively. The functionality offered by these object adapters has been replaced by that of the administrative facility, therefore these adapters (and their associated configuration properties) are deprecated.

Known Issues and Platform Notes

This page describes known issues and platform-specific notes for this Ice release.

On this page:

Java

Socket connection issue in Android emulator

An Ice application that attempts to connect to an invalid address can generate "spurious wakeup" messages in logcat when running under an emulator for Android 4.2 or later. This issue does not occur on hardware devices, however you can still experience lengthy delays before receiving an exception. As a defensive measure, it is a good idea to always set reasonable timeouts on your proxies to avoid unexpected delays.

Android Bluetooth limitations

A process can only establish one Bluetooth connection at a time to a particular remote endpoint. The process can have multiple connections open at the same time, but each of those connections must be to a different endpoint.

Anonymous Diffie Hellman ciphersuites

Recent versions of Java require low-strength ADH ciphersuites to be disabled when using TLS 1.0. It is no longer sufficient to use this IceSSL configuration:

```
IceSSL.Ciphers=NONE (DH_anon)
IceSSL.VerifyPeer=0
```

We recommend using this setting instead:

```
IceSSL.Ciphers=NONE (DH_anon.*AES)
IceSSL.VerifyPeer=0
```

Entropy pool causes hangs

When using the Ice for Java SSL plug-in (IceSSL), you may experience occasional hangs. The most likely reason is that your system's entropy pool is empty. If you have sufficient system privileges, you can solve this issue by editing the file *`java.home`*`/jre/lib/security/java.security` and changing it to use `/dev/urandom` instead of `/dev/random`. If you do not have permission to modify the security file, you can also use the command-line option shown below:

```
java -Djava.security.egd=file:/dev/urandom MyClass ...
```

IPv6 hang

On systems with IPv6 enabled, you may experience occasional hangs the first time an Ice object adapter is activated within a JVM. A work-around is to disable IPv6 support by setting the Java property `java.net.preferIPv4Stack` to true. For example:

```
java -Djava.net.preferIPv4Stack=true MyClass ...
```

Linux

Bluetooth limitations

A process can only establish one Bluetooth connection at a time to a particular remote endpoint. The process can have multiple connections open at the same time, but each of those connections must be to a different endpoint.

.NET Core IPv6 multicast

A bug in the .NET Core Socket implementation for Linux can cause a Socket exception when you set the interface endpoint option on a IPv6 multicast endpoint. Please refer to https://github.com/dotnet/corefx/issues/25525 for additional details.

.NET Core SSL/TLS certificate chains

A bug in the .NET Core `SslStream` implementation for Linux can cause a peer to receive a partial chain when the certificate chain contains intermediate certificates. Please refer to https://github.com/dotnet/corefx/issues/25581 for additional details.

Windows

Anti-Virus when testing WebSocket

If your anti-virus intercepts `http` traffic on localhost, a number of tests in the Ice test suite may fail due to timeouts when running the test suite with the `ws` (WebSocket) protocol. The work-around is to disable the "web access protection" feature of the anti-virus entirely or at least for localhost (127.0.0.1).

Multicast and Windows Store Apps

Network isolation for Windows Store Apps blocks multicast datagrams on the loopback interface. As a result, IceDiscovery and IceLocatorDiscovery won't be able to discover peers listening on the loopback interface.

TLS 1.2 with Windows 7, Windows 8.1 and Windows Server 2012

A bug in SChannel's implementation of TLS 1.2 that affects Windows versions prior to Windows 10 can result in SSL handshake failures when client and server negotiate a DHE-based cipher suite. Applications can work around this by disabling DHE cipher suites. See the links below for more information about this issue:

- https://connect.microsoft.com/IE/feedback/details/1253526/tls-serverkeyexchange-with-1024-dhe-may-encode-dh-y-as-127-bytes-breaking-internet-explorer-11
- https://github.com/dotnet/corefx/issues/7812#issuecomment-305848835

Using the Windows Binary Distributions

This page provides important information for users of the Ice binary distributions on Windows platforms.

On this page:

Overview of the Ice Binary Distributions for Windows

Ice 3.7 provides the following binary distributions for Windows:

- A traditional Windows installer file, ice-3.7.1.msi
- Several NuGet packages: zeroc.ice.v100, zeroc.ice.v120, zeroc.ice.v140, zeroc.ice.v141, zeroc.ice.uwp.v140, zeroc.ice.uwp.v141 and zeroc.ice.net

The table below shows which distribution(s) you should install depending on your needs.

Activity	Compiler or Environment	Distribution to Install	Files Installed
Develop C++ desktop applications	Visual Studio 2010 Visual Studio 2013 Visual Studio 2015 Visual Studio 2017	zeroc.ice.v100 zeroc.ice.v120 zeroc.ice.v140 zeroc.ice.v141	The complete Ice C++ SDK for the selected compiler, with x86, x64, Debug and Release binaries All Slice compilers (slice2cpp, slice2cs, slice2java, etc.) All Ice services (Glacier2, IceBridge, IceGrid, IcePatch2, IceStorm), and the associated command-line utility tools (icegridadmin, icestormadmin, etc.)
Develop C++/CX UWP applications	Visual Studio 2015 Visual Studio 2017	zeroc.ice.uwp.v140 zeroc.ice.uwp.v141	The complete Ice C++/CX SDK for the selected compiler, with x86, x64, Debug and Release binaries, and slice2cpp
Develop C# or other .NET applications	Visual Studio 2013, 2015 or 2017 .NET Core 2.0 SDK	zeroc.ice.net	The complete Ice C# /.NET SDK, including `slice2cs` for Windows.
Develop PHP applications	PHP 7.1	Ice MSI	The IceGrid GUI admin tool
Develop Java applications	JDK 8	Ice MSI	All the Slice compilers (slice2java, slice2php, etc.)
Administer IceGrid deployments		Ice MSI	All the Ice Slice files
Deploy Ice services (Glacier2, IceBridge, IceGrid, IcePatch2, IceStorm)		Ice MSI	The Ice for PHP extension (for PHP 7.1) All Ice services (Glacier2, IceBridge, IceGrid, IcePatch2, IceStorm), and the associated command-line utility tools (icegridadmin, icestormadmin, etc.) (x64 Release only)

If you are developing applications with Visual Studio, you should also install Ice Builder for Visual Studio.

C++ run time
Ice MSI
All the C++ binaries installed by the Ice MSI are x64 Release, built with Visual Studio 2015. The Ice MSI installs the corresponding Visual C++ run time on your computer: you don't need to install Visual Studio or any Visual Studio Redistributable to use this distribution.

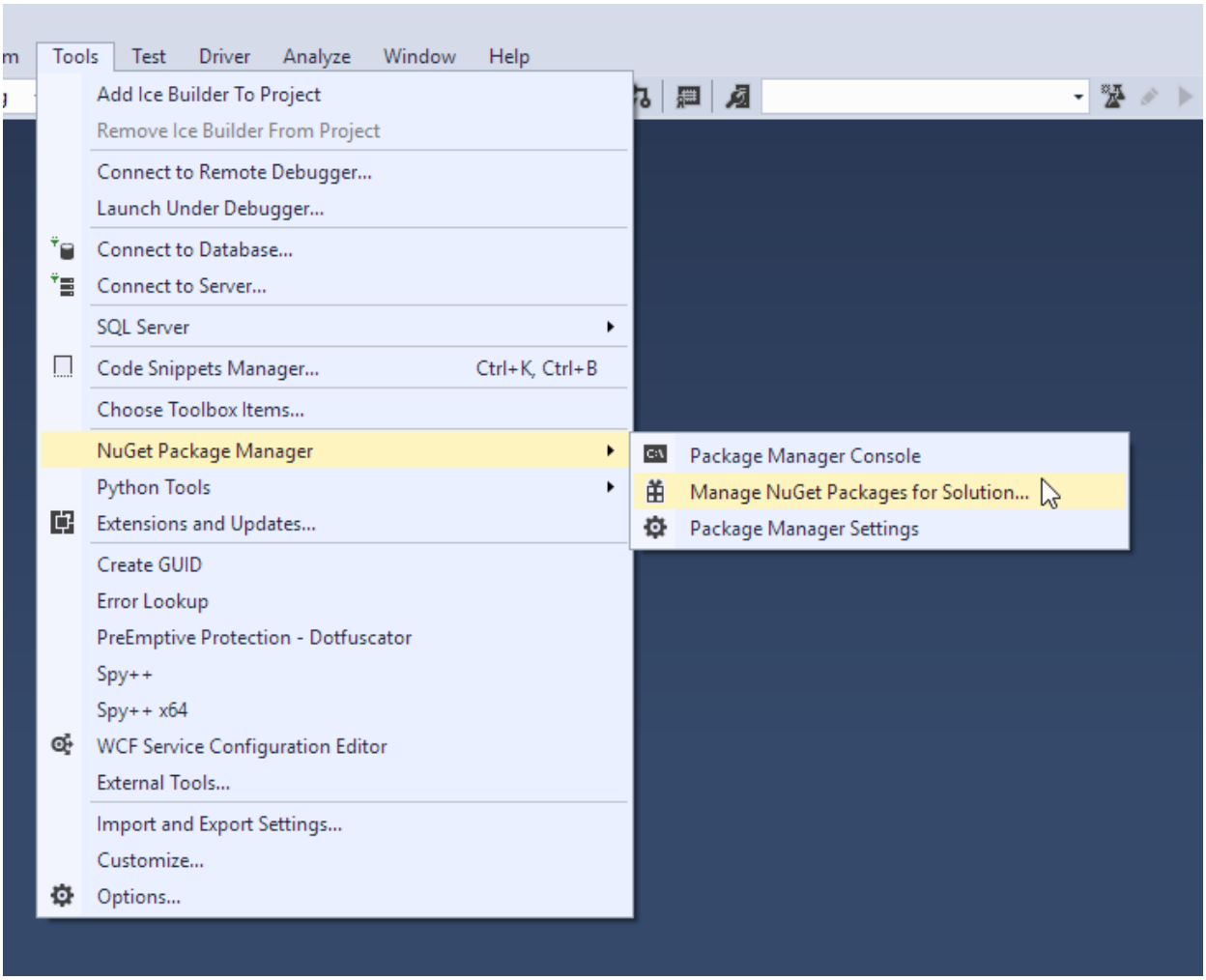
Ice NuGet packages
The Ice NuGet packages do not install the Visual C++ run time on your computer.

NuGet Package Installation

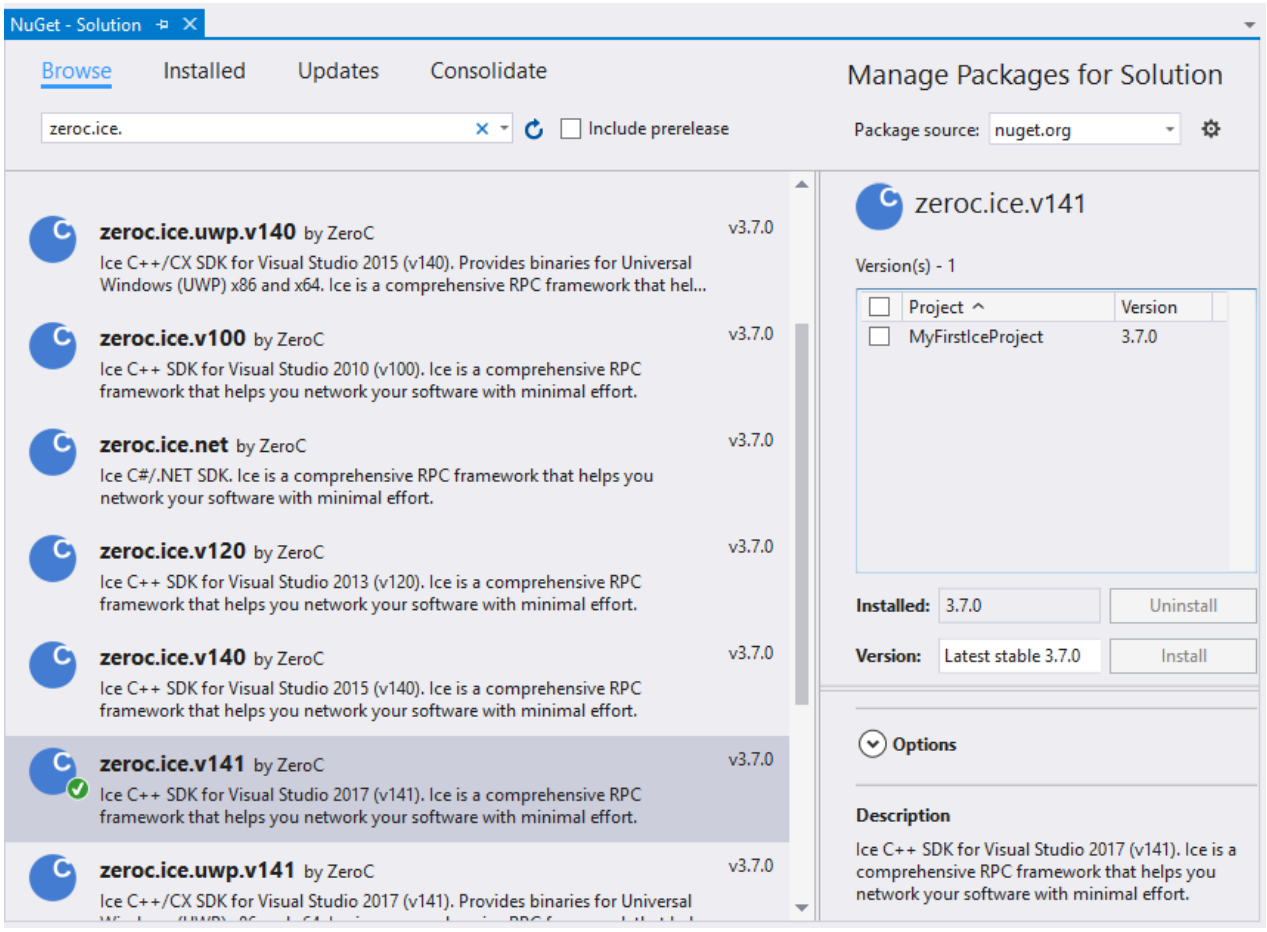
Adding a NuGet Package to a Visual Studio Project

Follow these steps to install Ice NuGet packages on your computer.

1. Open the Visual Studio Solution that contains the project you want to work on.
2. Open the NuGet Package Manager from the Tools menu:



3. On the next screen, select the zeroc.ice package you want to install, then select the project in which you want to install it, and finally click the Install button.
   NuGet will install this package in the `packages` folder next to your Solution file and configure the selected project to use it.



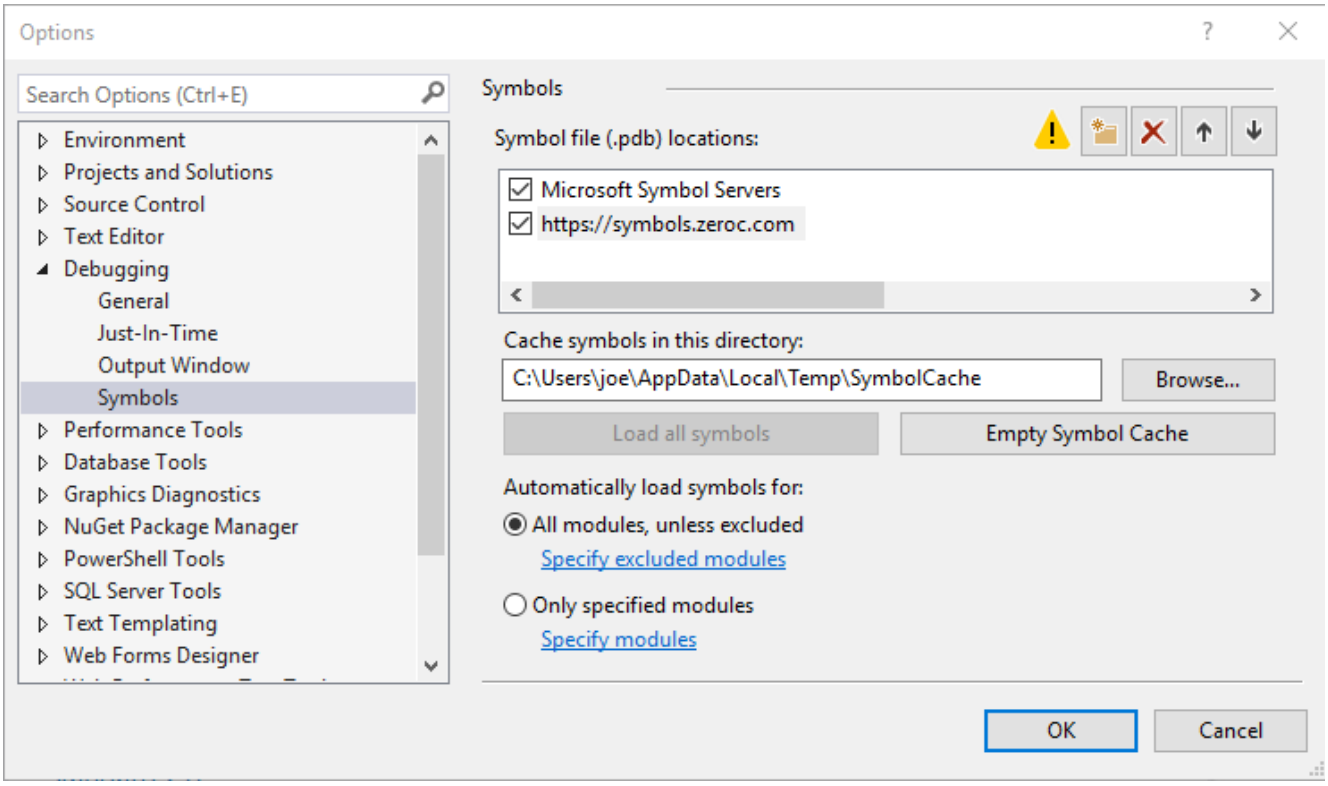
The screenshot above shows a solution with a C++ project. The process is the same with UWP and .NET projects.

If you are using Ice Builder, you also need to add `zeroc.icebuilder.msbuild` to your C++ or C# project.

ZeroC Symbol Server

The ZeroC symbol server, https://symbols.zeroc.com, provides debug symbols for the C++ binaries included in the NuGet packages published by ZeroC.

To use this symbol server, add its URL on Visual Studio's Symbols page:



Check "Enable source server support" in Visual Studio's debugging options to allow Visual Studio to fetch the corresponding source code.

The `zeroc.ice.net` and `zeroc.ice.uwp` NuGet packages include debug symbols (`pdb` files) and don't need this Symbol Server.

Debug Symbols and Stack Traces for C++ Applications

Ice C++ applications can print stack traces for Ice exceptions, which can be very helpful for debugging. In order to get usable stack traces, you need to install the corresponding Ice debug symbols in a local folder, and configure your system to look for debug symbols in this folder. We recommend you do the following:

1. Download and install the Debugging Tools for Windows.
2. Add these tools to your PATH, for example:

```
set PATH=C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64;%PATH%
```

3. Download the debug symbols of your Ice NuGet package(s) to your local `SymbolCache` folder, with the `symchk` tool included in the Debugging Tools for Windows. For example:

```
symchk /v /r packages\zeroc.ice.v140.3.7.1\build\native\bin\* /s
SRV*%TEMP%\SymbolCache*https://symbols.zeroc.com
```

(remove the `/v` option for a quieter output)
This copies debug symbols for all the binaries in the selected NuGet packages to `%TEMP%\SymbolCache`. `%TEMP%\SymbolCache` is also the default symbol cache folder for Visual Studio. This folder can cache symbols for any number of NuGet packages, in particular the debug symbols of different versions of the same package.

4. Set the environment variable _NT_SYMBOL_PATH to point to this local `SymbolCache` folder, for example:

```
set _NT_SYMBOL_PATH=%TEMP%\SymbolCache
```

Using the NuGet Packages

Information for C++ Developers

Once you've installed the Ice NuGet package into a C++ project as shown earlier, this project will find automatically all Ice C++ header files and import libraries. If you add `zeroc.icebuilder.msbuild` to this C++ project, Ice Builder will take care of compiling the Slice files in this project with slice2cpp (it uses the slice2cpp installed from the NuGet package).

Moreover, the Debugger Path is set and you can run your application directly from Visual Studio - there is no need to set any additional environment variables.

Occasionally, you may want to run your application outside Visual Studio, or use one of the Ice services or Ice tools included installed by the NuGet package. To do so, add the NuGet package's `bin` folder to your `PATH`.

The zeroc.ice.v<version> package is installed in the `packages` folder next to your Visual Studio solution file. For example, a Visual Studio 2017 solution would have a structure similar to the following:

```
C:\MyApplication\MyApplication.sln
C:\MyApplication\MyApplication\MyApplication.vcxproj
C:\MyApplication\packages\zeroc.ice.v141.3.7.1
```

With this example, you could set your PATH to:

```
set
PATH=C:\MyApplication\packages\zeroc.ice.v141.3.7.1\build\native\
x64\Release;%PATH%
```

Compiler Settings

Your application must be compiled with the same flags as the Ice libraries:

- Release: `/MD /EHsc`
- Debug: `/MDd /EHsc`

Add `ICE_CPP11_MAPPING` to the project preprocessor definitions if you want to use the Ice C++11 mapping. Without this definition, you will use the Ice C++98 mapping.

You don't need to list Ice import libraries such as `iceD.lib` when linking with Ice libraries. See Linking with C++ Libraries on Windows for additional details.

NuGet Package Details

The following table shows the Ice C++ NuGet package layout:

Folder	Description
`build\native\include`	C++ header files
`build\native\lib\<Platform>\<Configuration>`	C++ import libraries
`build\native\bin\<Platform>\<Configuration>`	C++ binaries (excluding Slice compilers)
`tools`	slice2xxx compilers
`build\native`	Visual Studio property and target files
`slice`	Slice files

Installing the NuGet package imports the property and target files from the `build\native` folder into the project. The property file defines properties used by Ice Builder; you can also use them in custom build steps.

The table below presents these properties:

Name	Value	Description
`IceVersion`	`3.7.1`	Ice string version
`IceIntVersion`	`30701`	Ice version as a numeric value
`IceVersionMM`	`3.7`	Major Minor version
`IceSoVersion`	`37`	Version used in dynamic libraries
`IceNugetPackageVersion`	`3.7.1`	NuGet package version
`IceHome`	`$(MSBuildThisFileDirectory)..\..`	Full path to the package root folder
`IceToolsPath`	`$(IceHome)\tools`	Full path to the folder of the Slice compilers

The targets file configures the C++ Additional Include Directories and Additional Library Directories to locate C++ headers and import libraries in the package's include and lib folders.

The NuGet package automatically adds its `build\native\lib\<Platform>\<Configuration>` folder to the `Additional Library Directories`, where `<Platform>` is the selected platform and `<Configuration>` is `Debug` when the MSBuild property `UseDebugLibraries` is `true`. Otherwise, `<Configuration>` is `Release`. Projects created with recent versions of Visual Studio set `UseDebugLibraries` to true automatically for debug projects (meaning projects that link with the Visual C++ debug run-time libraries); if you created your project with an older version of Visual Studio, you need to edit the project file and set `UseDebugLibraries` to `true` for your debug configurations, for example:

```
<PropertyGroup
Condition="'$(Configuration)|$(Platform)'=='Debug|x64'"
Label="Configuration">
   ...
   <UseDebugLibraries>true</UseDebugLibraries>
</PropertyGroup>
```

Information for C++/CX UWP Developers

Once you've installed the Ice NuGet package into a C++/CX project as shown earlier, this project will find automatically all Ice C++ header files and static libraries. If you add `zeroc.icebuilder.msbuild` to this C++ project, Ice Builder will take care of compiling the Slice files in this project with slice2cpp (it uses the slice2cpp installed from the NuGet package).

Compiler Settings

Ice for C++/CX UWP supports only the new C++11 mapping and requires that you add `ICE_CPP11_MAPPING` to your project's preprocessor definitions.

The Release versions of the Ice libraries are compiled with `/MD` to select the multi-threaded Visual C++ run-time library, while the Debug versions use `/MDd` to select the debug multi-threaded run-time library. Both versions of the Ice libraries are compiled with `/EHsc` to select an exception handling model.

Your application must be compiled with the same flags as the Ice libraries:

- Release: `/MD /EHsc /DICE_CPP11_MAPPING`
- Debug: `/MDd /EHsc /DICE_CPP11_MAPPING`

You don't need to list Ice import libraries such as `iceD.lib` when linking with Ice libraries. See Linking with C++ Libraries on Windows for additional details.

C++/CX SDK Nuget Package Details

The following table shows the Ice C++/CX NuGet package layout:

Folder	Description
`build\native\include`	C++ header files
`build\native\lib\<Platform>\<Configuration>`	C++ static libraries
`tools`	slice2cpp
`build\native`	Visual Studio property and target files
`slice`	Slice files

Installing the NuGet package imports the property and target files from the `build\native` folder into the project. The property file defines properties used by Ice Builder; you can also use them in custom build steps.

The table below presents these properties:

Name	Value	Description
`IceVersion`	`3.7.1`	Ice string version
`IceIntVersion`	`30701`	Ice version as numeric value
`IceVersionMM`	`3.7`	Major Minor version
`IceSoVersion`	`37`	Version used in dynamic libraries
`IceNugetPackageVersion`	`3.7.1`	NuGet package numeric version
`IceHome`	`$(MSBuildThisFileDirectory)..\..`	Full path to the package root folder
`IceToolsPath`	`$(IceHome)\tools`	Full path to the folder containing the Slice compilers.

The targets file configures the C++ Additional Include Directories and Additional Library Directories to locate C++ headers and import libraries in the package's `include` and `lib` folders.

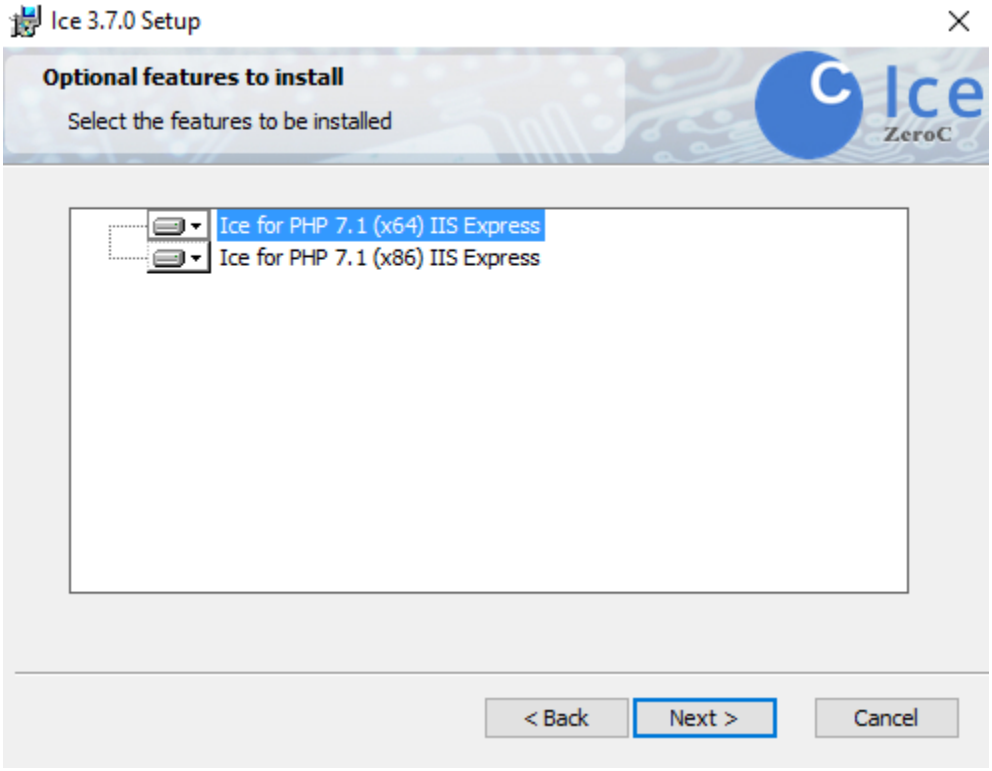
Information for C# and .NET Developers

Please refer to Building Ice Applications for .NET.

Using the Ice MSI Installation

Information for PHP Developers

The MSI distribution includes all the components required to develop Ice for PHP applications on Windows, including PHP and Slice source files, the slice2php compiler and the Ice for PHP extension. The MSI installer detects the PHP installations on your computer and allows you to select where to install the Ice for PHP extension:



For each PHP version you select, the installer copies the Ice for PHP extension to the `extensions` folder and updates the `php.ini` configuration file to load the extension; it also modifies `include_path` to include the Ice for PHP folder. The following lines are added to `php.ini`:

```
[PHP_ZEROC_ICE]
extension=php_ice_nts.dll
include_path=${include_path}";C:\Program Files\ZeroC\Ice-3.7.1\php"
```

The Ice MSI includes both x86 and x64 versions of the Ice for PHP extension built with the PHP 7.1 NTS libraries.

Configuration Files for IceGrid and Glacier2 Services

The `config` subdirectory of the Ice MSI installation includes sample configuration files for the Glacier2 router, IceGrid node, and IceGrid registry. These files provide a good starting point on which to base your own configurations, and they contain comments that describe the settings in detail.

The Ice manual provides more information on installing and running the IceGrid registry, IceGrid node, and Glacier2 router as Windows services.

Starting IceGrid GUI on Windows

You can launch IceGrid GUI using the shortcut that the Ice MSI installer created in your Start menu as **IceGrid GUI**. IceGrid GUI is a Java 8-based application.

Unattended Installation

The Ice MSI installer supports unattended installation. For example, in an administrative command window you can run:

```
start /wait Ice-3.7.1.msi /qn /l*v install.log
```

Windows may prompt you to confirm the installation, otherwise the installer runs using its default configuration (i.e., default installation folder, adds the installation's `bin` folder to the system PATH, and installs the PHP extension for the PHP installations it detects) but without any user interface. The installer will create a log of its activities in the file `install.log`.

Registry Key

The Ice MSI installer adds information to the Windows registry to indicate where it was installed. Developers can use this information to locate the Ice files in their applications.

The registration key used by this installer is:

`HKEY_LOCAL_MACHINE\SOFTWARE\ZeroC\Ice 3.7.1`

The install location is stored as a string value named `InstallDir`.

Using the Sample Programs on Windows

The Ice sample programs are provided in ice-demos GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages.

Clone the ice-demos repository as follows:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Using the Linux Binary Distributions

This page provides important information for users of the Ice binary distributions on Linux platforms. You can obtain these distributions at the ZeroC web site.

On this page:

Overview of the Ice Binary Distributions for Linux

DEB Packages

ZeroC provides the following DEB packages for Ubuntu:

Package	Description
zeroc-ice-all-dev	Meta package that installs all development packages
zeroc-ice-all-runtime	Meta package that installs all run-time packages, servers and utilities
libzeroc-ice-dev	C++ header files and libraries
libzeroc-ice3.7-java	Ice for Java JAR files
libzeroc-ice3.7	C++ run-time libraries
libzeroc-icestorm3.7	IceStorm service for IceBox C++
php-zeroc-ice	PHP extension and run-time files
python3-zeroc-ice	Python extension and run-time files
zeroc-glacier2	Glacier2 service
zeroc-icebox	IceBox server for C++
zeroc-icebridge	IceBridge service
zeroc-ice-compilers	Slice compilers, such as slice2cpp, slice2java and slice2php
zeroc-icegrid	IceGrid service
zeroc-icegridgui	IceGrid GUI application
zeroc-icepatch2	IcePatch2 service
zeroc-ice-slice	Slice files

`zeroc-ice-utils`	Utilities necessary for administering an Ice installation

RPM Packages

ZeroC provides the following RPMs for Red Hat Enterprise Linux, SUSE Linux Enterprise Server, and Amazon Linux:

RPM	Description
`ice-all-devel`	Meta package that installs all development packages
`ice-all-runtime`	Meta package that installs all run-time packages, servers and utilities
`glacier2`	Glacier2 service
`icebox`	IceBox server for C++
`icebridge`	IceBridge service
`icegrid`	IceGrid service
`icegridgui`	IceGrid GUI application
`icepatch2`	IcePatch2 executable
`ice-compilers`	Slice compilers, such as `slice2cpp`, `slice2java` and `slice2py`
`ice-slice`	Slice files
`ice-utils`	Utilities necessary for administering an Ice installation
`libice-c++-devel`	C++ header files and symbolic links to the C++ run-time libraries
`libice3.7-c++`	C++ run-time libraries
`libicestorm3.7`	IceStorm service
`php-ice`	PHP extension and run-time files
`python-ice`	Python extension and run-time files

ZeroC also supplies RPMs for the following third-party packages:

RPM	Description
`lmdb`	Admin tools for LMDB (statically linked)
`lmdb-devel`	Header file and static library for LMDB
`mcpp-devel`	Static library for the MCPP C++ preprocessor

> The RPM distribution no longer includes an RPM with the Ice for Java JAR files. See Building Ice Applications in Java for more information.

Bi-Arch Support on RHEL 7

On RHEL 7, all of the Ice packages listed above are provided for the x86_64 architecture, along with a limited subset of packages for the i686 architecture. The subset includes C++ run-time and development libraries, along with 32-bit versions of the IceBox server and IceStorm service. The 32-bit IceBox package installs the executables as `icebox32` (for the IceBox services built with the C++98 mapping) and `icebox32++11` (for IceBox services built with the C++11 mapping).

For development purposes, you will still need to install the 64-bit development kit packages: the 32-bit development kit packages complement these 64-bit packages.

Installing the Linux Distributions

This section describes how to install Ice binary packages for all of the supported Linux platforms.

Installing Ice on Ubuntu

Follow the instructions below to install Ice on Ubuntu.

1. Install ZeroC's key to avoid warnings with unsigned packages:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv
B6391CB2CFBA643D
```

2. Add the Ice repository to your system:

```
sudo apt-add-repository "deb
http://zeroc.com/download/Ice/3.7/ubuntu`lsb_release -rs` stable main"
```

3. Update the package list and install:

```
sudo apt-get update
sudo apt-get install zeroc-ice-all-runtime zeroc-ice-all-dev
```

Refer to the package summary if you would like to install fewer packages.

4. Install the source package (optional):

```
sudo apt-get source zeroc-ice3.7
```

Installing Ice on Red Hat Enterprise Linux 7

Follow the instructions below to install Ice on RHEL 7:

1. Add the Ice repositories to your system:

```
cd /etc/yum.repos.d
sudo wget https://zeroc.com/download/Ice/3.7/el7/zeroc-ice3.7.repo
```

2. Install Ice:

```
 sudo yum install ice-all-runtime ice-all-devel
```

Refer to the package summary if you would like to install fewer packages.

The yum documentation provides more information about installing packages on RHEL 7.

Installing Ice on Amazon Linux

Follow the instructions below to install Ice on Amazon Linux:

1. Add the Ice repositories to your system:

```
cd /etc/yum.repos.d
sudo wget https://zeroc.com/download/Ice/3.7/amzn1/zeroc-ice3.7.repo
```

2. Install Ice:

```
sudo yum install ice-all-runtime ice-all-devel
```

Refer to the package summary if you would like to install fewer packages.

The EC2 documentation provides more information about installing packages on Amazon Linux.

Installing Ice on SUSE Linux Enterprise Server 12

Follow the instructions below to install Ice on SLES 12:

1. Add the Ice repositories to your system and import the public key:

```
wget https://zeroc.com/download/Ice/3.7/sles12/zeroc-ice3.7.repo
sudo zypper addrepo zeroc-ice3.7.repo
sudo sudo rpm --import
https://zeroc.com/download/GPG-KEY-zeroc-release-B6391CB2CFBA643D
```

2. Install Ice:

```
sudo zypper install ice-all-runtime ice-all-devel
```

Refer to the package summary if you would like to install fewer packages.

Setting Up your Linux Environment to Use Ice

After installing Ice, read the relevant language-specific sections below to learn how to configure your environment and start programming with Ice.

C++

When compiling and linking Ice for C++ programs, you must pass the `-pthread` option. A typical compile command would look like this:
C++11 C++98

```
c++ -c -DICE_CPP11_MAPPING -pthread myprogram.cpp
```

```
c++ -c -pthread myprogram.cpp
```

C++11 and C++98 in the tabs above correspond to the Ice C++ mapping you're using.

When linking a program you must link with at least the Ice library. A typical link command would look like this:

C++11 C++98

```
c++ -o myprogram myprogram.o -pthread -lIce++11
```

```
c++ -o myprogram myprogram.o -pthread -lIce
```

Additional libraries are necessary if you are using an Ice service such as IceGrid or Glacier2.

PHP

The Ice extension for PHP is loaded automatically when the interpreter loads the contents of the file `/etc/php.d/ice.ini` (on Red Hat Enterprise Linux and Amazon Linux) or `/etc/php5/conf.d/ice.ini` (on SUSE Linux Enterprise Server and Ubuntu). This file contains the line shown below:

```
extension=ice.so
```

You can modify this file to include additional configuration directives.

At run time, the PHP interpreter requires the Ice shared libraries.

You can verify that the Ice extension is installed properly by examining the output of the `php -m` command, or by calling the `phpinfo()` function from a script.

Your application will also need to include at least some of the Ice for PHP run-time source files (installed in `/usr/share/php` on RHEL, Amazon Linux, and Ubuntu, and in `/usr/share/php5` on SLES). This installation directory is included in PHP's default include path, which you can verify by executing the following command:

```
php -i | grep include_path
```

If the installation directory is listed, no further action is necessary to make the run-time source files available to your application. Otherwise, you can modify the `include_path` setting in `php.ini` to add the installation directory:

```
include_path = /usr/share/php:...
```

Another option is to modify the include path from within your script prior to including any Ice run-time file:

```
ini_set('include_path', ini_get('include_path') . PATH_SEPARATOR .
'/usr/share/php');
require 'Ice.php'; // Load the core Ice run time definitions.
```

SELinux Notes (for Red Hat Enterprise Linux users)

SELinux augments the traditional Unix permissions with a number of new features. In particular, SELinux can prevent the `httpd` daemon from opening network connections and reading files without the proper SELinux types.

If you suspect that your PHP application does not work due to SELinux restrictions, we recommend that you first try it with SELinux disabled. As root, run:

```
setenforce 0
```

to disable SELinux until the next reboot of your computer.

If you want to run `httpd` with the Ice extension and SELinux enabled, you must do the following:

1. Allow httpd to open network connections:

   ```
   setsebool httpd_can_network_connect=1
   ```

   You can add the -P option to make this setting persistent across reboots.

2. Make sure any `.ice` file used by your PHP scripts can be read by `httpd`. The enclosing directory also needs to be accessible. For example:

   ```
   chcon -R -t httpd_sys_content_t /opt/MyApp/slice
   ```

For more information on SELinux in Red Hat Enterprise Linux, refer to this Red Hat document.

Using the Sample Programs on Linux

Sample programs for all programming languages are available in a separate ice-demos GitHub repository. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Starting IceGrid GUI on Linux

You can launch IceGrid GUI with the **`icegridgui`** command. IceGrid GUI is a Java 8-based application.

Startup Scripts for IceGrid and Glacier2 Services

All distributions include the following service configuration files:

- `/etc/icegridregistry.conf`
- `/etc/icegridnode.conf`
- `/etc/glacier2router.conf`

Distributions which use `sysvinit` or `upstart` contain the following sample scripts:

- /etc/init.d/icegridregistry
- /etc/init.d/icegridnode
- /etc/init.d/glacier2router

Distributions which use systemd contain the following services:

- icegridregistry.service
- icegridnode.service
- glacier2router.service

The installation also creates a user account and group for running these services (account ice and group ice), and data directories for icegridregistry and icegridnode (/var/lib/ice/icegrid/registry and /var/lib/ice/icegrid/node1).

By default, all these services are off at all runlevels. You need to manually switch on one or more runlevels, as shown below:

```
#
# On systems using svsvinit, configure the icegridregistry to start at
the
# default run levels:
#
sudo chkconfig icegridregistry on

#
# On systems using systemd, start icegridregistry with the multi-user
target
#
sudo systemctl enable icegridregistry.service
```

Before doing so, please review the script itself and its associated configuration file.

Using the macOS Binary Distribution

This page provides important information for users of the Ice binary distribution for macOS.

On this page:

Overview of the macOS Binary Distribution

Ice for C++, Java, Objective-C, and PHP on macOS are provided through the Homebrew formulas.

This `ice` formula includes the following components by default:

- Run-time libraries for C++ and Objective-C (macOS)
- Executables for IceGrid, IceStorm, Glacier2, IceBridge, and IcePatch2 services (macOS)
- Tools and libraries for developing Ice applications (macOS)
- Xcode SDKs for C++ and Objective-C (macOS, iOS and iPhone Simulator)

The `php-ice` formula includes:

- Run-time libraries for Homebrew's PHP (macOS)

Homebrew Taps

The `ice` formula is available in two taps:

- `homebrew/core`, the default homebrew tap
- `zeroc-ice/tap`, ZeroC's homebrew tap

The `php-ice` and `ice-builder-xcode` formulae are available in:

- `zeroc-ice/tap`, ZeroC's homebrew tap

> When a new version of Ice is released, `zeroc-ice/tap` provides the new release immediately. It can take several days (and occasionally more) for the `homebrew` taps to be updated.

Installing the macOS binary distribution

Using Homebrew, you can install the distribution with this command:

```
brew install zeroc-ice/tap/ice [--with-java]
[--with-additional-compilers] [--without-xcode-sdk]
```

The `--with-java` option builds the Java components and the IceGrid Admin app. You can also install IceGrid GUI on its own by downloading IceGrid GUI.dmg.

The `--with-additional-compilers` option installs `slice2py`, `slice2js`, and `slice2rb`.

The `--without-xcode-sdk` option skips the Xcode SDKs.

> If you want to install a binary bottle, as opposed to building everything from sources, do not specify any build option when installing `ice`.

You can install Ice for PHP with this command:

```
brew install zeroc-ice/tap/php-ice
```

Setting up your macOS environment to use Ice

After installing Ice, read the relevant language-specific sections below to learn how to configure your environment and start programming with Ice.

C++

Compiling and Linking

When compiling Ice for C++ programs, you must pass the `-pthread` option. A typical compile command would look like this:

C++11 C++98

```
c++ -c -DICE_CPP11_MAPPING -pthread myprogram.cpp
```

```
c++ -c -pthread myprogram.cpp
```

`C++11` and `C++98` in the tabs above correspond to the Ice C++ mapping you're using.

When linking a program you must link with at least the Ice library. A typical link command would look like this:

C++11 C++98

```
c++ -o myprogram myprogram.o -lIce++11
```

```
c++ -o myprogram myprogram.o -lIce
```

Additional libraries are necessary if you are using an Ice service such as IceGrid or Glacier2.

If you want to include Ice in your application bundle, you will need to copy the necessary Ice libraries to the `Contents/Frameworks` subdirectory of your bundle and use `@loader_path/../Frameworks` as the run path when linking the application.

Please refer to the `dyld` man page on your macOS system to learn more about `@loader_path`.

Objective-C

Compiling and Linking

When compiling Ice for Objective-C programs, you must pass the `-pthread` option. A typical compile command would look like this:

```
cc -c -pthread myprogram.m
```

When linking a program you must link with `libIceObjC`. A typical link command would look like this:

```
cc -o myprogram myprogram.o -lIceObjC -framework Foundation
```

Additional libraries are necessary if you are using an Ice service such as IceGrid or Glacier2.

If you want to include Ice in your application bundle, you will need to copy the necessary Ice libraries to the `Contents/Frameworks` subdirectory of your bundle and use `@loader_path/../Frameworks` as the run path when linking the application.

Please refer to the `dyld` man page on your macOS system to learn more about `@loader_path`.

Using Xcode SDKs

In order to use one of the Ice Xcode SDKs with your Xcode project, you need to:

- add `/usr/local/opt/ice/sdk/$(PLATFORM_NAME).sdk` to your *Additional SDKs*
- link your application with at least
  - `C++11: -lIce++11 -liconv -lbzip2`
  - `C++98: -lIce -liconv -lbzip2`
  - `Objective-C: -ObjC -lIce -lIceObjC -lc++ -liconv -lbzip2`
- On `iphoneos` and `iphonesimulator`, link your application with `CFNetwork.framework` and `UIKit.framework`

The following optional libraries are included in the Ice Xcode SDKs:

Name	Link your C++11 application with	Link your C++98 application with	Link your Objective-C application with	Additional dependencies
Glacier2 client library	`-lGlacier2++11`	`-lGlacier2`	`-lGlacier2ObjC`	
IceDiscovery plug-in	`-lIceDiscovery++11`	`-lIceDiscovery`	`-lIceDiscovery`	
IceGrid client library	`-lIceGrid++11`	`-lIceGrid`	`-lIceGridObjC`	
IceIAP plug-in (iphoneos and iphonesimulator only)	`-lIceIAP++11`	`-lIceIAP`	`-lIceIAP -lIceIAPObjC`	`ExternalAccessory.framewor`
IceLocatorDiscovery plug-in	`-lIceLocatorDiscovery++11`	`-lIceLocatorDiscovery`	`-lIceLocatorDiscovery`	
IceSSL plug-in	`-lIceSSL++11`	`-lIceSSL`	`-lIceSSL -lIceSSLObjC`	`Security.framework`
IceStorm client library	`-lIceStorm++11`	`-lIceStorm`	`-lIceStormObjC`	

> The Ice Xcode SDKs include only static libraries.

We also recommend installing the Ice Builder for Xcode, which helps you compile Slice files to C++ and Objective-C within Xcode . The builder is not included in the `ice` formula and must be installed separately with:

```
brew install zeroc-ice/tap/ice-builder-xcode
```

PHP

The Ice extension for PHP is loaded automatically when the interpreter loads the contents of the file `/usr/local/etc/php/{{version}}/php.d/ext-ice.ini` (generated on install). This file contains the following:

```
[ice]
extension="/usr/local/opt/php{{version}}-ice/ice.so"
include_path="/usr/local/opt/php{{version}}-ice"
```

At run time, the PHP interpreter requires the Ice shared libraries.

You can verify that the Ice extension is installed properly by examining the output of the `php -m` command, or by calling the `phpinfo()` function from a script.

Using the Sample Programs on macOS

Sample programs for all programming languages are available in a separate GitHub repository. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Starting IceGrid GUI on macOS

You can launch IceGrid GUI with the `IceGridGUI` application installed in your `/Applications` directory. IceGrid GUI is a Java 8-based application.

Building Ice Applications for .NET

This page provides important information for .NET developers.

On this page:

Building Ice Applications for .NET with Visual Studio

Install the following software and then refer to the Ice Builder for Visual Studio instructions:

1. A supported version of Visual Studio
   With Visual Studio 2017, you can optionally install the .NET Core cross-development toolset to create applications for .NET Core 2.0.
2. The Ice Builder for Visual Studio extension
3. The `zeroc.ice.net` NuGet package, described later on this page

Building Ice Applications for .NET with the .NET Core SDK

Install the following software and then refer to the Ice Builder for MSBuild instructions:

1. The .NET Core SDK for your operating system
2. The `zeroc.ice.net` NuGet package, described later on this page
3. The `slice2cs` compiler

`slice2cs` is a command-line tool written in C++ and available on most platforms

Platform	Distribution	Package with slice2cs
Ubuntu	apt packages	`zeroc-ice-compilers`
RHEL	RPMs	`ice-compilers`
Windows	NuGet	`zeroc.ice.net`

> On Windows, you can use Ice Builder for Visual Studio to configure Ice Builder for MSBuild.

Programming Language

You can use any .NET programming language with Ice, however, the preferred programming language for Ice .NET applications is C# since:

- the only Slice language mapping for .NET is Slice to C#
- the only Slice compiler for .NET, `slice2cs`, generates C# code
- Ice for .NET is itself written in C#

zeroc.ice.net NuGet Package

The Ice for .NET (zeroc.ice.net) NuGet package is organized as follows:

Folder	Contents
`lib\net45`	Assemblies for .NET Framework 4.5.1
`lib\netstandard2.0`	Assemblies for .NET Standard 2.0
`tools`	`slice2cs.exe`, `slice2html.exe` (Windows-only native tools)
`tools\net45`	`iceboxnet.exe` app for .NET Framework 4.5.1, `bzip2.dll` Windows x64 native library
`tools\netcoreapp2.0`	`iceboxnet.dll` app for .NET Core 2.0, `bzip2.dll` Windows x64 native library
`build`	MSBuild support files
`slice`	Slice files

Adding zeroc.ice.net to a Visual Studio or MSBuild project imports props files from the build folder into that project.

The props file defines the following properties:

Name	Value	Description
`IceVersion`	`3.7`	Ice string version
`IceIntVersion`	`30701`	Ice version as a numeric value
`IceVersionMM`	`3.7`	Major Minor version
`IceSoVersion`	`37`	Version used in dynamic libraries
`IceNugetPackageVersion`	`3.7.1`	Nuget package numeric version
`IceHome`	`$(MSBuildThisFileDirectory)..\..`	Full path to the package root folder
`IceToolsPath`	`$(IceHome)\tools`	Full path to the tools folder

These properties are used by Ice Builder, and you can also use them in custom build steps.

.NET Framework and .NET Standard Assemblies

`zeroc.ice.net` includes two sets of Ice assemblies: one set of assemblies for the .NET Framework 4.5 and another set for .NET Standard 2.0.

These assemblies are the same except for the differences described below:

	.NET Framework 4.5 Assemblies	.NET Standard 2.0 Assemblies
Run-time platform	Windows	Windows, Linux, macOS
Target Framework	.NET Framework 4.5.1 or greater on Windows	Any implementation of .NET Standard 2.0, including .NET Core 2.0 and .NET Framework 4.6.1.
Ice properties can be read from the Windows Registry		
Signals caught by Ice.Application	Signal catching implemented using the Windows native function SetConsoleCtrlHandler.	Signal catching implemented using the portable .NET event Console.KeyPress.

> The .NET Standard 2.0 assemblies are expected to work with any .NET implementation of .NET Standard 2.0, however, they are currently tested and supported only with .NET Core 2.0 on Windows and Linux.

Compression with bzip2

Ice for .NET supports the optional compression of Ice requests and responses using the bzip2 native library. The bzip2 native DLL for Windows x64 is included in the `zeroc.ice.net` package. You can use the bzip2 system library on Linux and macOS.

Using the Sample Programs

Sample programs are available at the ice-demos GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages. You can clone this repository with:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Building Ice Applications in Java

This page provides important information for Java developers.

On this page:

Prerequisites

In order to build applications with Ice in Java, you need:

1. the Ice JAR files (`ice.jar`, `icediscovery.jar`, `icegrid.jar`, etc.)
   These JAR files are provided through Maven, as described below. Ice binary distributions for Debian and Ubuntu also include these JAR files.

2. the `slice2java` compiler
   `slice2java` is a command-line tool written in C++ and available on most platforms

Platform	Distribution	Package with slice2java
Debian and Ubuntu	apt packages	zeroc-ice-compilers
RHEL, SLES, Amazon Linux	RPMs	ice-compilers
macOS	homebrew	always installed
Windows	MSI	always installed
Windows	NuGet	zeroc.ice.v100, zeroc.ice.v120, zeroc.ice.v140, zeroc.ice.v141

3. the Ice Builder for Gradle, if you are using Gradle.

Java and Java Compat Mappings

Ice provides two distinct Slice-to-Java mappings:

- Java
  This is a new mapping that takes advantage of features in Java 8. We recommend you select this mapping for new Ice-based applications written in Java.

- Java Compat
  This mapping is largely backward-compatible with prior Ice releases and does not depend on any Java 8-specific language or run-time features.

slice2java, the Slice-to-Java code generator, generates code for the Java mapping by default.

Maven Repository

You can fetch all Ice build artifacts from the Maven Central repository. ZeroC provides the following JAR files, all in group `com.zeroc` with version `3.7.1`:

Java Java Compat

Name	Description
`glacier2`	Generated proxy and skeleton classes plus helper classes for connecting to the Glacier2 service
`ice`	Ice core
`icebox`	The IceBox server, and generated proxy and skeleton classes for connecting to IceBox
`icebt`	The IceBT plug-in (only for Android)
`icediscovery`	The IceDiscovery plug-in
`icegrid`	Generated proxy and skeleton classes for connecting to the IceGrid service
`icelocatordiscovery`	The IceLocatorDiscovery plug-in
`icepatch2`	Generated proxy and skeleton classes for connecting to the IcePatch2 service
`icessl`	The IceSSL plug-in
`icestorm`	Generated proxy and skeleton classes for connecting to the IceStorm service

Name	Description
`glacier2-compat`	Generated proxy and skeleton classes plus helper classes for connecting to the Glacier2 service
`ice-compat`	Ice core and IceSSL plug-in
`icebox-compat`	The IceBox server, and generated proxy and skeleton classes for connecting to IceBox
`icebt-compat`	The IceBT plug-in (only for Android)
`icediscovery-compat`	The IceDiscovery plug-in
`icegrid-compat`	Generated proxy and skeleton classes for connecting to the IceGrid service
`icelocatordiscovery-compat`	The IceLocatorDiscovery plug-in

`icepatch2-compat`	Generated proxy and skeleton classes for connecting to the IcePatch2 service
`icestorm-compat`	Generated proxy and skeleton classes for connecting to the IceStorm service

Bzip2

Ice for Java supports protocol compression using the BZip2 implementation included with Apache Commons Compress. Compression is automatically enabled if these classes are present in your `CLASSPATH`. The Maven package id for the JAR file is as follows:

groupId	version	artifactId
org.apache.commons	1.14	commons-compress

These classes are a pure Java implementation of the bzip2 algorithm and therefore add significant latency to Ice requests.

Using the Sample Programs

Sample programs are available at the ice-demos GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages. You can clone this repository with:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Using Ice on Android

This page provides important information for users of the Ice for Java distribution.

On this page:

Overview of Ice on Android

Ice is available for Android through the Ice for Java distribution. To develop an Ice application for Android you will need the following components:

- Binary distribution of Ice for your development platform (Linux, macOS, Windows)
- Android Studio
- Ice Builder for Gradle

> You can use either the Java or the Java Compat mapping in your Android project. Note however that the Java mapping uses Java 8 language features and currently requires the Java 8 toolchain provided in Android Studio 3.

Configuring an Android Studio project

Android Studio uses a Gradle-based build system. Using Ice requires modifying your project's `build.gradle` settings file(s) to include the Ice JAR files. For convenience we recommend using the Ice Builder for Gradle plug-in to help automate the compilation of your Slice files.

The following snippets give an example of configuring the Ice and Ice Builder for Gradle sections for an Android project.

Ice build.gradle configuration

```
buildscript {
    repositories {
        mavenCentral()
        maven {
            url "https://plugins.gradle.org/m2/"
        }
    }
    dependencies {
        classpath
"gradle.plugin.com.zeroc.gradle.ice-builder:slice:1.4.5"
    }
}
apply plugin: 'java'
apply plugin: 'slice'
slice {
    java {
        srcDir = '.'
    }
}
dependencies {
    compile 'com.zeroc:ice:3.7.1'
}
```

Using the Sample Programs

Sample programs are available at the ice-demos GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

You can then import the directory `java/android` or `java-compat/android` into Android Studio.

Using Ice with Yocto

On this page:

Overview of Ice with Yocto

The Yocto Project allows you to create a custom Linux distribution for your embedded product. With ZeroC's meta layer for Yocto, you can easily include Ice for C++, Ice for Python and/or the Glacier2 service in your custom Linux distribution.

Development Boards

Ice was tested with the following boards and images:

Board	Yocto version
Freescale SABRE SD	2.4 (Rocko) with the Freescale Community BSP
BeagleBone Black board (Rev C)	2.4 (Rocko)

Meta Layer

Refer to the README.md file in the zeroc-ice/meta-zeroc repository.

Installing the ZeroC Meta Layer

Simply clone the `meta-zeroc` git repository and add `meta-zeroc` to the BBLAYERS variable in your `bblayers.conf` file:

In /home/user/repos

```
# Clone poky on branch rocko
git clone -b rocko git://git.yoctoproject.org/poky.git

# Clone meta-zeroc on branch rocko
git clone -b rocko git://github.com/zeroc-ice/meta-zeroc.git
```

bblayers.conf

```
BBLAYERS ?= " \
  /home/user/repos/poky/meta \
  /home/user/repos/poky/meta-yocto \
  /home/user/repos/poky/meta-yocto-bsp \
  /home/user/repos/meta-zeroc \
  "
```

The `meta-zeroc` repository has the same branches as the poky repository.

Using the ZeroC Meta Layer to install Ice

To include Ice in your image, add the package from the `zeroc-ice` recipe you wish to use to your `local.conf` file:

```
# Install Ice for C++ dynamic libraries and Ice for Python to image
IMAGE_INSTALL_append = " zeroc-ice zeroc-ice-python"

# Add the development package to the SDK
TOOLCHAIN_TARGET_TASK_append = " zeroc-ice-dev zeroc-ice-staticdev"

# Add the development package to the Native SDK
TOOLCHAIN_HOST_TASK_append  = " nativesdk-zeroc-ice-dev"
```

To ensure the correct Ice version is used you can set `PREFERRED_VERSION_zeroc-ice` in your local.conf file:

```
PREFERRED_VERSION_zeroc-ice = "3.7.1"
```

Setting up your cross development environment to use Ice

With the variables set in `local.conf` as shown above, you can generate an SDK for your image with the following command:

Create SDK for the core-image-minimal Image

```
bitbake core-image-minimal -c populate_sdk
```

Once complete, this SDK can be found in `tmp/deploy/sdk`. Execute the `.sh` file to install. By default your SDK will be installed into `/opt/poky/<version>/`. You can then source the cross-development environment as follows:

```
source
/opt/poky/2.4/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi
```

Using IceSSL with Platform CAs

To use the `IceSSL.UsePlatformCAs` property you will need to install the `ca-certificates` package as part of your image. Since OpenSSL does not have its default `CAfile` (where IceSSL looks for the default platform CAs) set to the certificate bundle installed from `ca-certifcates` you will need to set an additional IceSSL property:

```
IceSSL.CAs=/etc/ssl/certs/ca-certificates.crt
```

Using the Ice Sample Programs

Sample programs are available in the zeroc-ice/ice-demos repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

The Ice demos for C++ and Python are located in the `cpp11`, `cpp98` and `python` directories. Review the build instructions for more information.

Using the JavaScript Distribution

This page provides important information for users of the Ice for JavaScript distribution.

On this page:

Overview of the Ice for JavaScript Distribution

Ice for JavaScript is available as npm and bower packages. The following npm packages are provided for Node.js developers:

- ice - Ice for JavaScript run time
- slice2js - Slice-to-JavaScript compiler
- gulp-ice-builder - Gulp plug-in to integrate `slice2js` with the gulp build system

The following bower package is provided for developers of browser-based applications:

- bower-ice - Ice for JavaScript run time

Installing the Ice for JavaScript Distribution

Using NPM

Install the Ice for JavaScript run time libraries using `npm`:

```
npm install ice --save
```

This package also includes the browser version of Ice for JavaScript in `node_modules/lib`.

Using CDN

Ice for JavaScript browser libraries are available at https://cdnjs.com/libraries/ice/3.7.1

Slice-to-JavaScript Compiler

The Slice-to-JavaScript compiler (`slice2js`) can be installed with this command:

```
npm install --save-dev slice2js
```

This command builds the compiler from source therefore you must have a supported C++ compiler installed.

> In order to use `slice2js` on the command line, use the `npm --global` option to install this package.

The `slice2js` package includes all of the standard Slice files and automatically adds the slice directory to the include file search path.

Using Ice for JavaScript

The Ice manual provides a complete description of the JavaScript mapping.

Node.js Applications

You can use several top-level Ice packages in your Node.js applications, as shown below:

JavaScript

```
var Ice      = require('ice').Ice;
var Glacier2 = require('ice').Glacier2;
var IceStorm = require('ice').IceStorm;
var IceGrid  = require('ice').IceGrid;

var communicator = Ice.initialize(process.argv);
var proxy = communicator.stringToProxy("hello:tcp -h localhost -p
10000");
```

Browser Applications

Add the necessary `<script>` tags to your html to include the Ice for JavaScript components you require:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/Ice.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/Glacier2.js"></sc
ript>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/IceStorm.js"></sc
ript>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/IceGrid.js"></scr
ipt>
<script type="text/javascript">

var communicator = Ice.initialize();
var proxy = communicator.stringToProxy("hello:ws -h localhost -p
10002");
</script>
```

For older browsers that do not support all of the required EcmaScript 6 features used by Ice for JavaScript, we provide a pre-compiled version of the libraries using the Babel JavaScript compiler. These libraries depend on the babel polyfill run time and are available in the `es5` subdirectory with the same names as the main libraries:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/babel-polyfill/6.23.0/polyf
ill.min.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/es5/Ice.js"></scr
ipt>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/es5/Glacier2.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/es5/IceStorm.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/ice/3.7.1/es5/IceGrid.js"><
/script>
<script type="text/javascript">

var communicator = Ice.initialize();
var proxy = communicator.stringToProxy("hello:ws -h localhost -p
10002");
</script>
```

Minified versions are available with the `.min.js` extension.

Slice-to-JavaScript Compiler

You can execute the Slice-to-JavaScript compiler from code as shown below:

JavaScript

```
var slice2js = require('slice2js');
slice2js.compile(["Hello.ice"]);
```

If you installed the `slice2js` package globally, you can also run `slice2js` on the command line.

Refer to the manual for a description of the arguments accepted by `slice2js`.

Using the Sample Programs

The Ice sample programs are provided in a GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Using the MATLAB Distribution

This page provides important information for users of the Ice for MATLAB distribution.

On this page:

Overview of the Ice for MATLAB Distribution

Ice for MATLAB is available as a toolbox for supported versions of MATLAB. Each toolbox provides the following components:

- Ice library for MATLAB
- Standard Slice files
- Slice-to-MATLAB compiler (`slice2matlab`)

Installing the Ice for MATLAB Distribution

Download the toolbox file for your MATLAB version from https://zeroc.com/download/Ice/3.7. The file is named `ice-3.7.1-<version>.mltbx`, where `<version>` is your MATLAB version.

Now open MATLAB and navigate to the directory that contains the toolbox file. Double-click on the toolbox file to begin the installation.

Upon completion, MATLAB places the toolbox files in the Add-Ons directory specified in your Preferences settings. By default, this is a directory in your Documents folder.

You can manage the toolbox by choosing *Add-Ons* / *Manage Add-Ons*.

Using Ice for MATLAB

The installation process automatically appends the add-on directories to your MATLAB path. The installation includes a MATLAB script named `slice2matlab` that you can use to easily run the Slice-to-MATLAB compiler from the MATLAB console. To verify that Ice is installed, try running the script:

```
>> slice2matlab -h
```

The Ice manual provides a complete description of the MATLAB mapping, including the options for generating MATLAB code from Slice definitions.

At a minimum, your application will need to load the Ice library by calling `loadlibrary` as follows:

MATLAB

```
loadlibrary('ice', @iceproto)
```

Using the Sample Programs

Sample programs are provided in a separate GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Using the Python Distribution

This page provides important information for users of the Ice for Python distribution.

On this page:

Overview of the Ice for Python Distribution

Ice for Python is available as a collection of Python packages on the Python Package Index. Each package provides the following:

- Ice extension for Python
- Slice files for all Ice components (Glacier2, Ice, IceGrid etc.)
- Slice-to-Python compiler (`slice2py`)

> The binary distributions for Linux also include Python packages.

Installing the Ice for Python Distribution

Install Ice for Python using the pip source distribution. The following command will build and install the source package.:

```
pip install zeroc-ice
```

On Linux this command will build and install the source package. On macOS and Windows it installs a pre-built wheel for the following configurations:

- Python 2.7 on macOS (64 bit) and Windows (32 bit and 64 bit)
- Python 3.6 on Windows (32 bit and 64 bit)

Using Ice for Python

The installation process automatically adds the Ice modules to Python's package directory and adds the Slice-to-Python compiler (`slice2py`) to a directory that's likely already in your executable search path. To verify that Ice is installed, execute these commands:

```
python
>>> import Ice
>>> Ice.getSliceDir()
```

The output of `getSliceDir` shows where the Slice files have been installed.

The Ice manual provides a complete description of the Python mapping, including the options for generating Python code from Slice definitions.

Using the Sample Programs

Sample programs are provided in a separate GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Using the Ruby Distribution

This page provides important information for users of the Ice for Ruby distribution.

On this page:

Overview of the Ice for Ruby Distribution

Ice for Ruby is available as a collection of Ruby gems. Each package provides the following components:

- Ice extension for Ruby
- Standard Slice files
- Slice-to-Ruby compiler (`slice2rb`)

Installing the Ice for Ruby Distribution

Linux and macOS users can install Ice for Ruby using this command:

```
gem install zeroc-ice
```

This gem builds Ice for Ruby from source and supports Ruby 2.0 or later.

Using Ice for Ruby

The installation process automatically adds the Ice modules to Ruby's package directory and adds the Slice-to-Ruby compiler (`slice2rb`) to a directory that's likely already in your executable search path. To verify that Ice is installed, execute these commands:

```
irb
irb> require 'Ice'
irb> Ice::getSliceDir()
```

The output of `getSliceDir` shows where the standard Slice files have been installed.

The Ice manual provides a complete description of the Ruby mapping, including the options for generating Ruby code from Slice definitions.

Using the Sample Programs

Sample programs are provided in a separate GitHub repository. You can browse this repository to see build and usage instructions for all supported programming languages. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/ice-demos.git
cd ice-demos
```

Getting Started with Ice on AWS

This page explains how to get started with Ice on the Amazon Web Services (AWS) platform.

Select and Launch AMI

Ice 3.7 is available for AWS on the following AMIs

- **ZeroC Ice for Red Hat Enterprise Linux** - Red Hat Enterprise Linux 7.3
- **ZeroC Ice for Ubuntu** - Ubuntu 16.04 (LTS)

Navigate to the `EC2 Management Console` and follow these steps:

1. Select `Launch Instance`
2. Navigate to the AWS Marketplace page for the ZeroC Ice AMI you wish to use.
3. Choose the instance type you wish to run. ZeroC Ice AMIs are available are available in many sizes.
4. Configure your `Security Group` to allow access to port 22 for SSH login, as well as any ports for any Ice servers you plan on running. Glacier2 listens (typically) on port `4064` (ssl/tls). If you plan on using Glacier then you need to add these port(s) as well.
5. Select which `Key Pair` to use for access to the instance once it is running.
6. Configure any remaining instance settings as desired.
7. Click Accept Terms & Launch with 1-Click. Be sure to read the End-User License Agreement (EULA) beforehand.

Login

You can log in to your instance through SSH.

```
ssh -i <keypair> -l <username> <ip address>
```

Where,

- `<keypair>` - The SSH keypair selected when you launched your instance
- `<username>` - `ec2-user` for Red Hat based instance and `ubuntu` for Ubuntu based instance
- `<ip address>` - IP address of your instance

Package Configuration

ZeroC Ice AMIs have Ice 3.7 package repositories pre-configured. These repositories contain packages for both development and run time.

Red Hat Enterprise Linux

To install all development and run time packages:

```
sudo yum install ice-all-runtime ice-all-devel
```

See Using the Linux Binary Distributions for a list of packages.

Ubuntu

To install all development and run time packages:

```
sudo apt-get install zeroc-ice-all-runtime zeroc-ice-all-dev
```

See Using the Linux Binary Distributions for a list of packages.