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
- Ice Objects
- Proxies
- Stringified Proxies
- Direct Proxies
- Indirect Proxies
- Direct Versus Indirect Binding
- Fixed Proxies
- Routed Proxies
- Replication
- Replica Groups
- Servants
- At-Most-Once Semantics
- Synchronous Method Invocation
- Asynchronous Method Invocation
- Asynchronous Method Dispatch
- Oneway Method Invocation
- Batched Oneway Method Invocation
- Datagram Invocations
- Batched Datagram Invocations
- Run-Time Exceptions
- User Exceptions
- Properties

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* systems.

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. 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)*: clients can invoke operations *asynchronously*, that is, the client uses a proxy as usual to invoke an operation but, in addition to passing the normal parameters, also passes a *callback object* and the client invocation returns immediately. Once the operation completes, the client-side run time invokes a method on the callback object passed initially, passing the results of the operation to the callback object (or, in case of failure, passing exception information).

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.

Another way to use asynchronous method dispatch is to complete an operation, so the results of the operation are returned to the client, but to keep the execution thread of the operation beyond the duration of the operation invocation. This allows you to continue processing after results have been returned to the client, for example, to perform cleanup or write updates to persistent storage.

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
- Locators
- Object Life Cycle
- Bidirectional Connections

- Glacier2
- IceStorm