Protocol Messages

Previous Next

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.

Back to Top ^

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`.

Back to Top ^

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`.

Back to Top ^

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: Unknown macro: 'table' 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.

Back to Top ^

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`.

Back to Top ^

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.

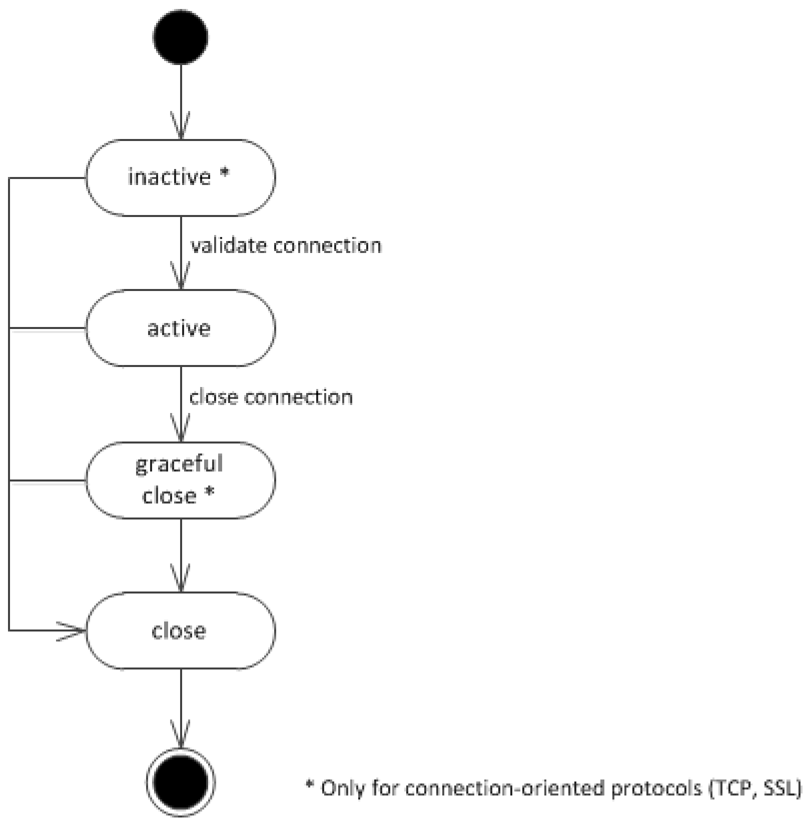
The server side can close a connection whenever no operation invocation is in progress that was invoked via that connection. This guarantees that the server will not violate at-most-once semantics: an operation, once invoked in a servant, is allowed to complete and its results are returned to the client. 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.

Back to Top ^

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.

Back to Top ^

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).

Back to Top ^

See Also

- Data Encoding
- Protocol Compression
- Object Identity
- Versioning
- Request Contexts
- Oneway Invocations
- Batched Invocations
- Protocol and Encoding Versions
- Active Connection Management
- Automatic Retries
- Connection Closure

Previous

Next