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 is possible for the client to lose already-buffered oneway requests. We therefore recommend that you disable active connection management for the server side if clients use oneway (or batched oneway) requests. 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++

```
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`.

- 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++

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a oneway proxy.
//
Ice::ObjectPrx oneway;
try {
    oneway = o->ice_oneway();
} catch (const Ice::NoEndpointException&) {
    cerr << "No endpoint for oneway invocations" << endl;
}

// 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++

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Safe down-cast to actual type.
//
PersonPrx person = PersonPrx::checkedCast(o);

if (person) {
    // Get a oneway proxy.
    //
    PersonPrx onewayPerson;
    try {
        onewayPerson = person->ice_oneway();
    } catch (const Ice::NoEndpointException&) {
        cerr << "No endpoint for oneway invocations" << endl;
    }

    // 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.