Datagram Invocations

Previous Next

On this page:

- Design Considerations for Datagram Invocations
- Creating Datagram Proxies
- Using UDP Multicast

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.)

Back to Top ^

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

```
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;
}
```

C++98

```
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.

Back to Top ^

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.

Back to Top ^

See Also

- Terminology
- Oneway Invocations
- The Ice Protocol

Previous

Next