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.

Back to Top ^

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.

Back to Top ^

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
```

Back to Top ^

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
```

Back to Top ^

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.

Back to Top ^