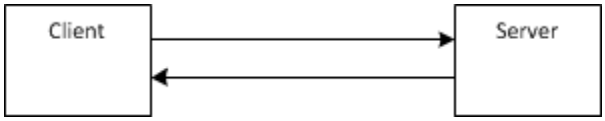
# Bidirectional Connections

On this page:



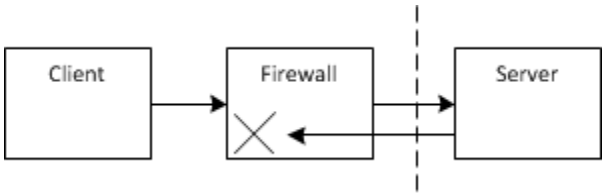
## Use Cases for Bidirectional Connections

An Ice connection normally allows requests to flow in only one direction. If an application's design requires the server to make callbacks to a client, the server usually establishes a new connection to that client in order to send callback requests, as shown below:




*Callbacks in an open network.*

Unfortunately, network restrictions often prevent a server from being able to create a separate connection to the client, such as when the client resides behind a firewall as shown here:




*Callbacks with a firewall.*

In this scenario, the firewall blocks any attempt to establish a connection directly to the client.

For situations such as these, a bidirectional connection offers a solution. Requests may flow in both directions over a bidirectional connection, enabling a server to send callback requests to a client over the client's existing connection to the server.

There are two ways to make use of a bidirectional connection. First, you can use a Glacier2 router, in which case bidirectional connections are used automatically. If you do not require the functionality offered by Glacier2 or you do not want an intermediary service between clients and servers, you can configure bidirectional connections manually.

The remainder of this section discusses manual configuration of bidirectional connections. An example that demonstrates how to configure a bidirectional connection is provided in the directory `demo/Ice/bidir` of your Ice distribution.

## Configuring a Client for Bidirectional Connections

A client needs to perform the following steps in order to configure a bidirectional connection:

1. Create an object adapter to receive callback requests. This adapter does not require a name or endpoints if its only purpose is to receive callbacks over bidirectional connections.
2. Register the callback object with the object adapter.
3. Activate the object adapter.
4. Obtain a connection object by calling `ice_getConnection` on the proxy.
5. Invoke `setAdapter` on the connection, passing the callback object adapter. This associates an object adapter with the connection and enables callback requests to be dispatched.
6. Pass the identity of the callback object to the server.

The C++ code below illustrates these steps:

**C++**

```
Ice::ObjectAdapterPtr adapter = communicator->createObjectAdapter("");
Ice::Identity ident;
ident.name = IceUtil::generateUUID();
ident.category = "";
CallbackPtr cb = new CallbackI;
adapter->add(cb, ident);
adapter->activate();
proxy->ice_getConnection()->setAdapter(adapter);
proxy->addClient(ident);
```

The last step may seem unusual because a client would typically pass a proxy to the server, not just an identity. For example, you might be tempted to give the proxy returned by the adapter operation `add` to the server, but this would not have the desired effect: if the callback object adapter is configured with endpoints, the server would attempt to establish a separate connection to one of those endpoints, which defeats the purpose of a bidirectional connection. It is just as likely that the callback object adapter has no endpoints, in which case the proxy is of no use to the server.

Similarly, you might try invoking `createProxy` on the connection to obtain a proxy that the server can use for callbacks. This too would fail, because the proxy returned by the connection is for local use only and cannot be used by another process.

As you will see below, the server must create its own callback proxy.

### Disabling Active Connection Management

Active Connection Management (ACM) automatically and transparently closes idle connections. By default, ACM is enabled for client (outgoing) connections and disabled for server (incoming) connections. As far as the client-side Ice run time is concerned, a bidirectional connection is a client connection, therefore the client-side property `Ice.ACM.Client` governs ACM behavior for bidirectional connections.

In general, it is necessary to disable client-side ACM for bidirectional connections: since the outgoing connection is the only channel on which the server can send callback invocations to the client, allowing ACM to prematurely close the connection introduces the risk that the client might unknowingly fail to receive callbacks. To disable client-side ACM, set the property to zero:

```
Ice.ACM.Client=0
```

It is not necessary to disable client-side ACM if your client sends invocations at regular intervals, in which case Ice will never consider the connection to be idle and therefore ACM will not close it. The default value for `Ice.ACM.Client` is 60, meaning a connection that has been idle for 60 seconds is eligible to be closed.

## Configuring a Server for Bidirectional Connections

A server needs to take the following steps in order to make callbacks over a bidirectional connection:

1. Obtain the identity of the callback object, which is typically supplied by the client.
2. Create a proxy for the callback object by calling `createProxy` on the connection. The connection object is accessible as a member of the `Ice::Current` parameter to an operation implementation.

These steps are illustrated in the C++ code below:

**C++**

```
void addClient(const Ice::Identity& ident, const Ice::Current& curr)
{
    CallbackPrx client = CallbackPrx::uncheckedCast(curr.con->createProxy(ident));
    client->notify();
}
```

## Fixed Proxies

The proxy returned by a connection's `createProxy` operation is called a *fixed proxy*. It can only be used in the server process and cannot be marshaled or stringified by `proxyToString`; attempts to do so raise `FixedProxyException`.

A fixed proxy is bound to the connection that created it, and ceases to work once that connection is closed. If the connection is closed prematurely, either by active connection management (ACM) or by explicit action on the part of the application, the server can no longer make callback requests using that proxy. Any attempt to use the proxy again usually results in a `CloseConnectionException`.

Many aspects of a fixed proxy cannot be changed. For example, it is not possible to change the proxy's endpoints or timeout. Attempting to invoke a method such as `ice_timeout` on a fixed proxy raises `FixedProxyException`.

# Limitations of Bidirectional Connections

Bidirectional connections have certain limitations:

- They can only be configured for connection-oriented transports such as TCP and SSL.
- Most proxy factory methods are not relevant for a proxy created by a connection's `createProxy` operation. The proxy is bound to an existing connection, therefore the proxy reflects the connection's configuration. Attempting to change settings such as the proxy's timeout value causes the Ice run time to raise `FixedProxyException`. Note however that it is legal to configure a fixed proxy for using oneway or twoway invocations. You may also invoke `ice_secure` on a fixed proxy if its security configuration is important; a fixed proxy configured for secure communication raises `NoEndpointException` on the first invocation if the connection is not secure.
- A connection established from a Glacier2 router to a server is not configured for bidirectional use. Only the connection from a client to the router is bidirectional. However, the client must not attempt to manually configure a bidirectional connection to a router, as this is handled internally by the Ice run time.
- Bidirectional connections are not compatible with ACM.

# Threading Considerations for Bidirectional Connections

The Ice run time normally creates two thread pools for processing network traffic on connections: the client thread pool manages outgoing connections and the server thread pool manages incoming connections. All of the object adapters in a server share the same thread pool by default, but an object adapter can also be configured to have its own thread pool. The default size of the client and server thread pools is one.

The client thread pool processes replies to pending requests. When a client configures an outgoing connection for bidirectional requests, the client thread pool also becomes responsible for dispatching callback requests received over that connection.

Similarly, the server thread pool normally dispatches requests from clients. If a server uses a bidirectional connection to send callback requests, then the server thread pool must also process the replies to those requests.

You must increase the size of the appropriate thread pool if you need the ability to dispatch multiple requests in parallel, or if you need to make nested twoway invocations. For example, a client that receives a callback request over a bidirectional connection and makes nested invocations must increase the size of the *client* thread pool.

## See Also

- Glacier2
- Creating an Object Adapter
- Servant Activation and Deactivation
- Using Connections
- Object Identity
- Active Connection Management
- Proxy Methods
- Nested Invocations
- The Ice Threading Model
- Object Adapter Thread Pools