

Using Connections

Applications can gain access to an Ice object representing an established connection.

On this page:

- [The Connection Interface](#)
 - [Flushing Batch Requests for a Connection](#)
- [The Endpoint Interface](#)
 - [Opaque Endpoints](#)
- [Client-Side Connection Usage](#)
- [Server-Side Connection Usage](#)
- [Closing a Connection](#)
 - [Graceful Closure](#)
 - [Forceful Closure](#)

The Connection Interface

The Slice definition of the Connection interface is shown below:

Slice

```
module Ice {
    local class ConnectionInfo {
        bool incoming;
        string adapterName;
    };

    local interface Connection {
        void close(bool force);
        Object* createProxy(Identity id);
        void setAdapter(ObjectAdapter adapter);
        ObjectAdapter getAdapter();
        Endpoint getEndpoint();
        void flushBatchRequests();
        string type();
        int timeout();
        string toString();
        ConnectionInfo getInfo();
    };

    local class IPConnectionInfo extends ConnectionInfo {
        string localAddress;
        int localPort;
        string remoteAddress;
        int remotePort;
    };

    local class TCPConnectionInfo extends IPConnectionInfo {};

    local class UDPConnectionInfo extends IPConnectionInfo {
        string mcastAddress;
        int mcastPort;
    };
};

module IceSSL {
    local class ConnectionInfo extends Ice::IPConnectionInfo {
        string cipher;
        Ice::StringSeq certs;
    };
};
```

As indicated in the Slice definition, a connection is a [local interface](#), similar to a communicator or an object adapter. A connection therefore is only usable within the process and cannot be accessed remotely.

The `Connection` interface supports the following operations:

- `void close(bool force)`
Explicitly [closes the connection](#). The connection is closed gracefully if `force` is false, otherwise the connection is closed forcefully.
- `Object* createProxy(Identity id)`
Creates a special proxy that only uses this connection. This operation is primarily used for [bidirectional connections](#).
- `void setAdapter(ObjectAdapter adapter)`
Associates this connection with an object adapter to enable a [bidirectional connection](#).
- `ObjectAdapter getAdapter()`
Returns the object adapter associated with this connection, or nil if no association has been made.
- `Endpoint getEndpoint()`
Returns an [Endpoint object](#).
- `void flushBatchRequests()`
Flushes any pending [batch requests](#) for this connection.
- `string type()`
Returns the connection type as a string, such as "tcp".
- `int timeout()`
Returns the [timeout](#) value used when the connection was established.
- `string toString()`
Returns a readable description of the connection.
- `ConnectionInfo getInfo()`
This operation returns a `ConnectionInfo` class defined as follows:

Slice

```
local class ConnectionInfo {
    bool incoming;
    string adapterName;
};
```

The `incoming` member is true if the connection is an incoming connection and false, otherwise. If `incoming` is true, `adapterName` provides the name of the object adapter that accepted the connection. Note that the object returned by `getInfo` implements a more derived interface, depending on the connection type. You can down-cast the returned class instance and access the connection-specific information according to the type of the connection.

Flushing Batch Requests for a Connection

The `flushBatchRequests` operation blocks the calling thread until any batch requests that are queued for a connection have been successfully written to the local transport. To avoid the risk of blocking, you can also invoke this operation asynchronously using the `begin_flushBatchRequests` method (in those language mappings that support it).

Since batch requests are inherently oneway invocations, the `begin_flushBatchRequests` method does not support a request callback. However, you can use the exception callback to handle any errors that might occur while flushing, and the `sent` callback to receive notification that the batch request has been flushed successfully.

For example, the code below demonstrates how to flush batch requests asynchronously in C++:

C++

```

class FlushCallback : public IceUtil::Shared
{
public:

    void exception(const Ice::Exception& ex)
    {
        cout << "Flush failed: " << ex << endl;
    }

    void sent(bool sentSynchronously)
    {
        cout << "Batch sent!" << endl;
    }
};
typedef IceUtil::Handle<FlushCallback> FlushCallbackPtr;

void flushConnection(const Ice::ConnectionPtr& conn)
{
    FlushCallbackPtr f = new FlushCallback;
    Ice::Callback_Connection_flushBatchRequestsPtr cb =
        Ice::newCallback_Connection_flushBatchRequests(
            f, &FlushCallback::exception, &FlushCallback::sent);
    conn->begin_flushBatchRequests(cb);
}

```

For more information on asynchronous invocations, please see the relevant language mapping chapter.

The Endpoint Interface

The `Connection::getEndpoint` operation returns an interface of type `Endpoint`:

Slice

```

module Ice {
    const short TCPEndpointType = 1;
    const short UDPEndpointType = 3;

    local class EndpointInfo {
        int timeout;
        bool compress;
        short type();
        bool datagram();
        bool secure();
    };

    local interface Endpoint {
        EndpointInfo getInfo();
        string toString();
    };

    local class IPEndpointInfo extends EndpointInfo {
        string host;
        int port;
    };

    local class TCPEndpointInfo extends IPEndpointInfo {};

    local class UDPEndpointInfo extends IPEndpointInfo {
        byte protocolMajor;
        byte protocolMinor;
        byte encodingMajor;
        byte encodingMinor;
        string mcastInterface;
        int mcastTtl;
    };

    local class OpaqueEndpointInfo extends EndpointInfo {
        Ice::ByteSeq rawBytes;
    };
};

module IceSSL {
    const short EndpointType = 2;

    local class EndpointInfo extends Ice::IPEndpointInfo {};
};

```

The `getInfo` operation returns an `EndpointInfo` instance. Note that the object returned by `getInfo` implements a more derived interface, depending on the endpoint type. You can down-cast the returned class instance and access the endpoint-specific information according to the type of the endpoint, as returned by the `type` operation.

The `timeout` member provides the timeout in milliseconds. The `compress` member is true if the endpoint uses [compression](#) (if available). The `datagram` operation returns true if the endpoint is for a [datagram](#) transport, and the `secure` operation returns true if the endpoint uses [SSL](#).

The derived classes provide further detail about the endpoint according to its type.

Opaque Endpoints

An application may receive a proxy that contains an endpoint whose type is unrecognized by the Ice run time. In this situation, Ice preserves the endpoint in its encoded (*opaque*) form so that the proxy remains intact, but Ice ignores the endpoint for all connection-related activities. Preserving the endpoint allows an application to later forward that proxy with all of its original endpoints to a different program that might support the endpoint type in question.

Although a connection will never return an opaque endpoint, it is possible for a program to encounter an opaque endpoint when iterating over the endpoints returned by the [proxy method](#) `ice_getEndpoints`.

As a practical example, consider a program for which the [IceSSL](#) plug-in is not configured. If this program receives a proxy containing an SSL endpoint, Ice treats it as an opaque endpoint such that calling `getInfo` on the endpoint object returns an instance of `OpaqueEndpointInfo`.

Note that the `type` operation of the `OpaqueEndpointInfo` object returns the *actual* type of the endpoint. For example, the operation returns the value 2 if the object encodes an SSL endpoint. As a result, your program cannot assume that an `EndpointInfo` object whose type is 2 can be safely down-cast to `IceSSL::EndpointInfo`; if the `IceSSL` plug-in is not configured, such a down-cast will fail because the object is actually an instance of `OpaqueEndpointInfo`.

Client-Side Connection Usage

Clients obtain a connection by using one of the [proxy methods](#) `ice_getConnection` or `ice_getCachedConnection`. If the proxy does not yet have a connection, the `ice_getConnection` method immediately attempts to establish one. As a result, the caller must be prepared to handle [connection failure](#) exceptions. Furthermore, if the proxy denotes a [collocated object](#) and collocation optimization is enabled, calling `ice_getConnection` results in a `CollocationOptimizationException`.

If you wish to obtain the proxy's connection without the potential for triggering connection establishment, call `ice_getCachedConnection`; this method returns null if the proxy is not currently associated with a connection or if connection caching is disabled for the proxy.

As an example, the C++ code below illustrates how to obtain a connection from a proxy and print its type:

C++

```
Ice::ObjectPrx proxy = ...
try
{
    Ice::ConnectionPtr conn = proxy->ice_getConnection();
    cout << conn->type() << endl;
}
catch(const Ice::CollocationOptimizationException&)
{
    cout << "collocated" << endl;
}
```

Server-Side Connection Usage

Servers can access a connection via the `con` member of the `Ice::Current` parameter passed to every operation. For collocated invocations, `con` has a nil value.

For example, this Java code shows how to invoke `toString` on the connection:

Java

```
public int add(int a, int b, Ice.Current curr)
{
    if (curr.con != null)
    {
        System.out.println("Request received on connection:\n" + curr.con.toString());
    }
    else
    {
        System.out.println("collocated invocation");
    }
    return a + b;
}
```

Although the mapping for the Slice operation `toString` results in a Java method named `_toString`, the Ice run time implements `toString` to return the same value.

Closing a Connection

Applications should rarely need to close a connection explicitly, but those that do must be aware of its implications. Since there are two ways to close a connection, we discuss them separately.

Graceful Closure

Passing an argument of `false` to the `close` operation initiates graceful connection closure, as discussed in [Connection Closure](#). The operation blocks until all pending outgoing requests on the connection have completed.

Forceful Closure

A forceful closure is initiated by passing an argument of `true` to the `close` operation, causing the peer to receive a `ConnectionLostException`.

A client must use caution when forcefully closing a connection. Any outgoing requests that are pending on the connection when `close` is invoked will fail with a `ForcedCloseConnectionException`. Furthermore, requests that fail with this exception are not automatically retried.

In a server context, forceful closure can be useful as a defense against hostile clients.

The Ice run time interprets a `CloseConnectionException` to mean that it is safe to [retry](#) the request without violating at-most-once semantics. If automatic retries are enabled, a client must only initiate a graceful close when it knows that there are no outgoing requests in progress on that connection, or that any pending requests can be safely retried.

See Also

- [The Current Object](#)
- [Automatic Retries](#)
- [Connection Establishment](#)
- [Connection Closure](#)
- [Bidirectional Connections](#)
- [IceSSL](#)