Using Connections

Previous Next

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

On this page:

The `Connection` Interface

The Slice definition of the `Connection` interface is shown below:

Slice

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

        ["delegate"]
    local interface CloseCallback
    {
        void closed(Connection con);
    }

        ["delegate"]
    local interface HeartbeatCallback
    {
        void heartbeat(Connection con);
    }

    local enum ACMClose
    {
        CloseOff,
        CloseOnIdle,
        CloseOnInvocation,
        CloseOnInvocationAndIdle,
        CloseOnIdleForceful
    }

    local enum ACMHeartbeat
    {
        HeartbeatOff,
        HeartbeatOnInvocation,
        HeartbeatOnIdle,
        HeartbeatAlways
    }

    local struct ACM
```

```
    {
        int timeout;
        ACMClose close;
        ACMHeartbeat heartbeat;
    }

    local enum ConnectionClose
    {
        Forcefully,
        Gracefully,
        GracefullyWithWait
    }

    local interface Connection
    {
        void close(ConnectionClose mode);
        Object* createProxy(Identity id);
        void setAdapter(ObjectAdapter adapter);
        ObjectAdapter getAdapter();
        Endpoint getEndpoint();
        void flushBatchRequests();
        void setCloseCallback(CloseCallback callback);
        void setHeartbeatCallback(HeartbeatCallback callback);
        void setACM(optional(1) int timeout, optional(2) ACMClose close, optional(3) ACMHeartbeat heartbeat);
        ACM getACM();
        string type();
        int timeout();
        string toString();
        ConnectionInfo getInfo();
        void setBufferSize(int rcvSize, int sndSize);
        void throwException();
    }

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

    local class TCPConnectionInfo extends IPConnectionInfo
    {
        int rcvSize;
        int sndSize;
    }

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

    dictionary<string, string> HeaderDict;

    local class WSConnectionInfo extends ConnectionInfo
    {
            HeaderDict headers;
    }
}

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

```
}

module IceBT
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string localAddress = "";
        int localChannel = -1;
        string remoteAddress = "";
        int remoteChannel = -1;
        string uuid = "";
    }
}

module IceIAP
{
    local class ConnectionInfo extends Ice::ConnectionInfo
    {
        string name;
        string manufacturer;
        string modelNumber;
        string firmwareRevision;
        string hardwareRevision;
        string protocol;
    }
}
```

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(ConnectionClose mode)`
  Explicitly closes the connection using the given closure mode.
- `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.
- `void setCloseCallback(CloseCallback callback)`
  Associates a callback with this connection that is invoked whenever the connection is closed. Passing a nil value clears the current callback.
- `void setHeartbeatCallback(HeartbeatCallback callback)`
  Associates a callback with this connection that is invoked whenever the connection receives a heartbeat message. Passing a nil value clears the current callback.
- `void setACM(optional(1) int timeout, optional(2) ACMClose close, optional(3) ACMHeartbeat heartbeat)`
  Configures Active Connection Management settings for this connection. All arguments are optional, therefore you can change some of the settings while leaving the others unaffected. Refer to your language mapping for more details on optional parameters.
- `ACM getACM()`
  Returns the connection's current settings for Active Connection Management.
- `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` instance. . 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.

  The `incoming` member of the `ConnectionInfo` instance 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. The `connectionId` member contains the identifier set with the proxy `ice_connectionId` method.

  The `underlying` member contains the underlying transport information if the connection uses a transport that delegates to an underlying transport. For example, the SSL transport delegates to the TCP transport so the `underlying` data member of an SSL connection information is set to a `TCPConnectionInfo` instance. For a WSS connection, `getInfo` returns a `WSConnectionInfo` instance whose `underlying` data member is set to an `IceSSL::ConnectionInfo` whose `underlying` data member is set to a `TCPConnectionInfo`.

- `void setBufferSize(int rcvSize, int sndSize)`
  Sets the connection buffer receive and send sizes.

- `void throwException()`
  Throws an exception indicating the reason for connection closure. For example, `CloseConnectionException` is raised if the connection was closed gracefully by the remote peer, whereas `ConnectionManuallyClosedException` is raised if the connection was manually closed locally by the application. This operation does nothing if the connection is not yet closed.

Back to Top ^

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.

Since batch requests are inherently oneway invocations, the async `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++11

```
void flushConnection(Ice::CompressBatch compressBatch, const std::shared_ptr<Ice::Connection>& conn)
{
    // std::future version also available
    conn->flushBatchRequestsAsync(compressBatch,
                                  [](std::exception_ptr) { cout << "Flush failed" << endl; },
                                  [](bool) { cout << "Batch sent" << endl; });
}
```

C++98

```
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(Ice::CompressBatch compressBatch, 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(compressBatch, cb);
}
```

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

Back to Top ^

The `Endpoint` Interface

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

Slice

```
module Ice
{
    const short TCPEndpointType = 1;
    const short UDPEndpointType = 3;
    const short WSEndpointType = 4;
    const short WSSEndpointType = 5;
    const short BTEndpointType = 6;
    const short BTSEndpointType = 7;
    const short iAPEndpointType = 8;
    const short iAPSEndpointType = 9;


    local class EndpointInfo
    {
        EndpointInfo underlying;
        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;
        string sourceAddress;
    }

    local class TCPEndpointInfo extends IPEndpointInfo {};

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

    local class WSEndpointInfo extends EndpointInfo
    {
        string resource;
    }

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

module IceSSL
{
    local class EndpointInfo extends Ice::EndpointInfo {};
}


module IceBT
{
    local class EndpointInfo extends Ice::EndpointInfo
    {
        string addr;
        string uuid;
    }
}

module IceIAP
{
    local class EndpointInfo extends Ice::EndpointInfo
    {
        string manufacturer;
        string modelNumber;
        string name;
        string protocol;
    }
}
```

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 `underlying` member contains the underlying endpoint information if the transport delegates to an underlying transport. For example, the SSL transport uses the TCP transport so the `underlying` data member of an SSL endpoint information is set to a `TCPEndpointInfo` instance. For a WSS endpoint, `getInfo` returns a `WSEndpointInfo` instance whose `underlying` data member is set to an `IceSSL::EndpointInfo` whose `underlying` data member is set to a `TCPEndpointInfo`.

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

Back to Top ^

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`.

Back to Top ^

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 for this method to block and raise connection failure exceptions. (Use the asynchronous version of this method to avoid blocking.) If the proxy denotes a collocated object and collocation optimization is enabled, calling `ice_getConnection` returns null.

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++11

```
auto proxy = ...
try
{
    auto conn = proxy->ice_getConnection();
    if(conn)
    {
        cout << conn->type() << endl;
    }
    else
    {
        cout << "collocated" << endl;
    }
}
catch(const Ice::LocalException& ex)
{
    cout << ex << endl;
}
```

C++98

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

Back to Top ^

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, Current current)
{
    if(current.con != null)
    {
        System.out.println("Request received on connection:\n" + current.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.

Back to Top ^

Closing a Connection

Applications should rarely need to close a connection manually, but those that do must be aware of its implications. The `ConnectionClose` enumeration defines three closure modes:

- Forcefully
- Gracefully
- Gracefully with wait

Forcefully

A forceful closure causes 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 `ConnectionManuallyClosedException`. 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.

Gracefully

This mode initiates graceful connection closure and causes the local Ice run time to send a `CloseConnection` message to the peer. Any outgoing requests that are pending on the connection when `close` is invoked will fail with a `ConnectionManuallyClosedException`. Furthermore, requests that fail with this exception are not automatically retried.

In a server context, closing a connection gracefully causes Ice to discard any subsequent incoming requests; these requests should eventually be retried automatically when the client receives a `CloseConnection` message. The Ice run time in the server does not send the `CloseConnection` message until all pending dispatched requests have completed.

> After invoking `close(CloseGracefully)`, Ice considers the connection to be in a *closing* state until the remote peer completes its part of the graceful connection closure process. The connection could remain in this state for some time if the peer has no thread pool threads available to process the `CloseConnection` message, and this can prevent operations such as `Communicator::destroy` from completing in a timely manner. Ice uses a timeout to limit the amount of time it waits for a connection to be closed properly. Refer to `Ice.Override.CloseTimeout` for more information.

Gracefully with Wait

In a client context, this mode waits until all pending requests complete before initiating graceful closure. The call to `close` can block indefinitely until the pending requests have completed.

In a server context, closing a connection gracefully causes Ice to discard any subsequent incoming requests; these requests should eventually be retried automatically when the client receives a `CloseConnection` message. The Ice run time in the server does not send the `CloseConnection` message until all pending dispatched requests have completed.

> After invoking `close(CloseGracefullyWithWait)`, Ice considers the connection to be in a *closing* state until the remote peer completes its part of the graceful connection closure process. The connection could remain in this state for some time if the peer has no thread pool threads available to process the `CloseConnection` message, and this can prevent operations such as `Communicator::destroy` from completing in a timely manner. Ice uses a timeout to limit the amount of time it waits for a connection to be closed properly. Refer to `Ice.Override.CloseTimeout` for more information.

Back to Top ^

See Also

- The Current Object
- Automatic Retries
- Connection Establishment
- Connection Closure
- Bidirectional Connections
- IceSSL
- IceBT
- IceIAP

Previous

Next