Connection Timeouts

Previous Next

On this page:

Overview of Connection Timeouts

> Connection timeouts only affect network operations. Use invocation timeouts to limit the amount of time a client waits for an operation to complete.

Connection timeouts allow applications to detect low-level network problems in a reasonable period of time. Ice enforces connection timeouts when performing network operations such as establishing a connection, reading and writing to a connection, and closing a connection. Disabling connection timeouts means an application may not discover a network issue until much later (if at all) when low-level network protocols finally detect and report the problem, therefore Ice enables connection timeouts by default and we strongly encourage you to use them.

You should normally choose your connection timeouts based on the speed of the network on which the connections take place. For example, the connection timeout for a local gigabit network will usually be much smaller than the timeout for a 56kbps connection.

Finally, note that timeouts in Ice are "soft" timeouts, in the sense that they are not precise, real-time timeouts. (The precision is limited by the capabilities of the underlying operating system.)

Back to Top ^

Configuring Connection Timeouts

Ice supports several configuration properties that you can use to control connection timeouts:

- `Ice.Default.Timeout`
  This property specifies the default timeout in milliseconds for all connections. This property's default value of `60000` means connection timeouts are enabled by default with a timeout of 60 seconds.
- `Ice.Override.ConnectTimeout`
  This setting overrides any existing connection timeout, but only applies while Ice establishes a connection.
- `Ice.Override.CloseTimeout`
  This setting overrides any existing connection timeout, but only applies while Ice closes a connection.
- `Ice.Override.Timeout`
  This setting overrides any existing connection timeout. It applies to all network operations unless superseded by `Ice.Override.ConnectTimeout` or `Ice.Override.CloseTimeout`.

You can also configure connection timeouts individually for the endpoints of a proxy. Consider this example:

```
MyProxy=hello:tcp -h 10.0.0.1 -t 1000:tcp -h 205.125.53.4 -t 5000
```

This stringified proxy contains two TCP endpoints. The first endpoint refers to a host in a local network and its `-t` option specifies a connection timeout of one second. The second endpoint refers to a host on the internet and uses a connection timeout of five seconds. If a proxy's endpoints don't include timeouts, the endpoints inherit the communicator's default timeout set by `Ice.Default.Timeout`. The various `Ice.Override` properties take precedence over any endpoint timeouts.

Finally, the proxy method `ice_timeout` returns a new proxy in which all of its endpoints have the specified timeout:

C++11

```
auto p = communicator->stringToProxy("hello:tcp -h 10.0.0.1 -t 1000:tcp -h 205.125.53.4 -t 5000");

cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t 1000:tcp -h 205.125.53.4 -t 5000
p = p->ice_timeout(1500);
cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t 1500:tcp -h 205.125.53.4 -t 1500
```

C++98

```
Ice::ObjectPrx p = communicator->stringToProxy("hello:tcp -h 10.0.0.1 -t 1000:tcp -h 205.125.53.4 -t 5000");

cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t 1000:tcp -h 205.125.53.4 -t 5000
p = p->ice_timeout(1500);
cout << p->ice_toString() << endl; // Output: hello:tcp -h 10.0.0.1 -t 1500:tcp -h 205.125.53.4 -t 1500
```

The `Ice.Override` properties also take precedence over any timeouts configured via `ice_timeout`.

Back to Top ^

Connection Timeout Failures

Ice considers a connection timeout to indicate a serious error with the underlying network connection, causing Ice to immediately close the connection and report a `TimeoutException`, or one of its subclasses `ConnectTimeoutException` or `CloseTimeoutException` if appropriate, for all of the invocations pending on that connection.

> The application may not receive an exception immediately after a connection timeout occurs, as Ice may attempt the invocation again if automatic retries are enabled and the invocation qualifies for retry. Since its default configuration allows Ice to perform one retry without a delay, it will appear to the application as if the timeout takes twice as long as expected to be reported.

In most cases you won't need to handle a timeout as a special case and can use the typical exception clauses:

C++11

```
std::shared_ptr<Filesystem::FilePrx> myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

C++98

```
Filesystem::FilePrx myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

If your application needs to treat timeouts differently, you can catch them as follows:

C++11

```
std::shared_ptr<Filesystem::FilePrx> myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::ConnectTimeoutException&)
{
    cerr << "connect timed out!" << endl;
}
catch(const Ice::CloseTimeoutException&)
{
    cerr << "close timed out!" << endl;
}
catch(const Ice::TimeoutException&)
{
    cerr << "timed out!" << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

C++98

```
Filesystem::FilePrx myFile = ...;
myFile = myFile->ice_timeout(2500);

try
{
    Lines text = myFile->read();
}
catch(const Filesystem::GenericError& ex)
{
    cerr << ex.reason << endl;
}
catch(const Ice::ConnectTimeoutException&)
{
    cerr << "connect timed out!" << endl;
}
catch(const Ice::CloseTimeoutException&)
{
    cerr << "close timed out!" << endl;
}
catch(const Ice::TimeoutException&)
{
    cerr << "timed out!" << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "failed: " << ex << endl;
}
```

Back to Top ^

ACM and Timeouts

The Active Connection Management features can also help your application in detecting and dealing with connectivity issues and complement the connection timeout behavior. For example, connection timeouts only have an effect when the Ice run time is actively performing network operations, but it's also possible for problems to arise during periods of inactivity. These problems may not be noticed for some time.

The ACM heartbeat facility can optionally be configured to send heartbeat messages at regular intervals when a connection would otherwise be idle. Doing so makes it more likely that the Ice run time will detect a connectivity issue in a timely manner.

Back to Top ^

Connection Reuse and Timeouts

Ice tries to reuse existing connections as much as possible unless the application indicates otherwise. When an application makes its first invocation on a proxy, Ice searches its list of open connections to see if any of them match any of the proxy's endpoints. Determining whether an existing connection is "compatible" with a proxy endpoint, and therefore suitable for reuse, involves several criteria. One attribute that Ice considers is the timeout: the endpoint's timeout and the connection's timeout must match for the connection to be considered eligible for reuse.

If you encounter a situation where Ice is opening a new connection when you expect it to use an existing connection, your timeout configuration may be the reason.

> See Connection Establishment for more information on how Ice decides whether to open a new connection or reuse an existing one.

Back to Top ^

See Also

- Proxy Methods
- Invocation Timeouts
- Automatic Retries
- Connection Establishment

Previous

Next