Automatic Retries

Previous Next

Ice may automatically retry a proxy invocation after a failure. This is a powerful feature that, when used in the proper situations, can significantly improve the robustness of your application without any additional programming effort. The retry facility is governed by one overriding principle: always respect at-most-once semantics. At-most-once semantics dictate that the Ice run time in the client must never retry a failed proxy invocation unless Ice guarantees that the server has not already received the request, or unless the application declares that it is safe for Ice to violate at-most-once semantics for the request.

To understand the importance of obeying at-most-once semantics, consider the following Slice definition:

Slice

```
interface Account
{
    long withdraw(long amount);
}
```

The `withdraw` operation removes funds from an account. If an invocation of `withdraw` fails, automatically retrying the request introduces the risk of a duplicate withdrawal unless Ice is absolutely sure that the server has not already executed the request.

This page examines automatic retries in more detail.

On this page:

Automatic Retries for Request Failures

Ice considers a request to have failed if any of the following conditions are true:

- A connection could not be established
- A connection was lost before the reply was received
- A timeout expired
- An exception occurred while sending the request or receiving the reply
- An error occurred in the server while dispatching the request that causes the server to return an `UnknownException` or `RequestFailedException`

Ice considers an invocation that results in a user exception to be successful and therefore excludes it from consideration for automatic retries.

Ice must determine the answers to several questions to decide whether to retry a failed request:

1. What kind of error caused the request to fail?

Ice does not bother retrying a request if it knows the same error is going to occur again. For example, Ice never retries an invocation that raises a `MarshalException`, which indicates that there was a problem while encoding or decoding a message. Retrying such an invocation is unlikely to change the outcome. It also doesn't retry invocation timeouts, if a server didn't respond within the invocation timeout period, it's unlikely that a retry would provide better results.

Ice also never retries exceptions that derive from `RequestFailedException` because they indicate a permanent failure. One such subclass is `OperationNotExistException`, whose occurrence signals a serious problem in the application. For instance, it might mean that the client and server are using incompatible Slice definitions, or that the client is trying to invoke operations on the wrong object. The exception to this rule is `ObjectNotExistException`, which Ice does consider to be worthy of retry if the proxy in question is indirect because it gives an application the ability to transparently migrate an Ice object.

In addition to user exceptions and subclasses of `RequestFailedException`, a server can also return an instance of `UnknownException`, `UnknownLocalException`, or `UnknownUserException` to indicate that it encountered an unexpected exception while dispatching the request. These exceptions *are* eligible for retry.

2. When did the error occur?

If the error is still a candidate for retry, Ice needs to know whether the server has received the request. Naturally, the Ice run time in the client cannot possibly know that information until the server confirms it by sending a reply. However, to be conservative Ice assumes that the server has received the request as soon as Ice has written the *entire* protocol message to the client's local transport buffers. If the error occurred before Ice managed to write the complete message, retrying the request would not violate at-most-once semantics.

The Ice run time in the server also has the ability to notify the client that a request was not dispatched and therefore that it is safe for the Ice run time in the client to retry the request without violating at-most-once semantics. For example, this situation can occur when the server is shutting down while there are pending requests that have yet to be executed. Sending this notification allows a client to transparently fail over to another server.

3. Does the application require strict adherence to at-most-once semantics for this request?

An application can grant permission for Ice to violate at-most-once semantics for certain Slice operations by marking them as *idempotent*, causing Ice to retry a request that otherwise would be ineligible because the server has already received it. We discuss idempotent operations in more detail below.

If Ice determines that an invocation cannot be retried, it raises the exception that caused the request failure to the application. On the other hand, if Ice does retry the invocation and the subsequent retries also fail, Ice raises the *last* exception to the application. For example, if the first attempt fails with `ConnectionRefusedException` and the retry fails with `ConnectTimeoutException`, the invocation raises `ConnectTimeoutException` to the application.

Back to Top ^

Automatic Retries for Idempotent Operations

Annotating a Slice operation with the `idempotent` keyword notifies Ice that it can safely violate at-most-once semantics:

Slice

```
interface Account
{
    long withdraw(long amount);
    idempotent long getBalance();
}
```

Although `withdraw` clearly requires the stricter treatment, there is no harm in automatically retrying the `getBalance` operation even if the server executes the same request more than once.

In general, "read-only" operations are good candidates for the `idempotent` keyword whereas many mutating operations are not. However, the risk of duplicate requests is acceptable even for some kinds of mutating operations:

Slice

```
interface Account
{
    long withdraw(long amount);
    idempotent long getBalance();
    idempotent void changeAddress(string newAddress);
}
```

Here we have marked `changeAddress` as idempotent because executing the request twice has the same effect as executing it only once.

The benefit of the `idempotent` keyword and the associated relaxation of retry semantics is that an invocation that otherwise might have raised an exception has at least one more chance to succeed. Furthermore, the application does not need to initiate the retry, and in fact the retry activities are completely transparent: if a subsequent retry succeeds, the application receives its results as if nothing went wrong. The invocation only raises an exception once Ice has reached its configured retry limits.

Back to Top ^

Configuring Automatic Retries

Retry Intervals

The `Ice.RetryIntervals` property configures the retry behavior for a communicator and affects invocations on every proxy created by that communicator. (Retry behavior cannot be configured on a per-proxy basis.) The value of this property consists of a series of integers separated by whitespace. The number of integers determines how many retry attempts Ice makes, and the value of each entry represents a delay in milliseconds. If this property is not defined, the default behavior is to retry once immediately after the first failure, which is equivalent to the following property definition:

```
Ice.RetryIntervals=0
```

You may want a more elaborate configuration for your application, such as a gradual increase in the delay between retries:

```
Ice.RetryIntervals=0 100 500 1000
```

With this setting, Ice retries immediately as in the default case. If the first retry attempt also fails, Ice waits 100 milliseconds before trying again, then 500 milliseconds, and finally tries one more time after waiting one second.

In some situations you may need to disable retries completely. For example, an application might implement its own retry logic and therefore require immediate notification when a failure occurs. Clients that establish a session with a Glacier2 router also need to disable retries. To prevent automatic retries, use a value of `-1`:

```
Ice.RetryIntervals=-1
```

Back to Top ^

Retry Logging

To monitor Ice's retry activities, configure your program with the property `Ice.Trace.Retry` set to a non-zero value:

```
Ice.Trace.Retry=1
```

When retry tracing is enabled, Ice logs a message each time it attempts a retry; the log message includes a description of the exception that prompted the retry. Ice also logs a message when it reaches the retry limit.

You can configure Ice to log even more information about retries by setting the property to 2:

```
Ice.Trace.Retry=2
```

This setting prompts Ice to include additional details about connections and endpoints.

Back to Top ^

Timeouts and Automatic Retries

Ice does not retry an invocation that fails with an invocation timeout. However, an invocation that fails with a connection timeout can be eligible for retry.

If you test connection timeouts (for example, by attempting to connect to an unreachable IP address), you may notice that the `TimeoutException` is not raised as quickly as you would expect . Automatic retries are usually the reason for this situation.

For example, suppose a proxy is configured with a ten seconds connection timeout and automatic retries are enabled with the default setting (one immediate retry). If an invocation on that proxy fails due to a connection timeout and Ice determines that the invocation is eligible for retry (using the criteria described above), Ice immediately tries the invocation again and waits for another connection timeout period to expire before finally raising `TimeoutException`. From the application's perspective, the invocation fails after approximately twenty seconds.

Consequently, you can compute an approximate worst-case connection timeout value as follows, assuming the proxy has a single endpoint:

```
T = t * (N + 1) + D
```

where *t* is the connection timeout value, *N* is the number of retry intervals, and *D* is the sum of the retry intervals (the total delay between retries). Consider our example again:

```
Ice.RetryIntervals=0 10000 20000 30000
```

Using this configuration with a ten second connection timeout, our approximate worst-case timeout is `10 * 5 + 60 = 110` seconds.

Back to Top ^

Connections and Automatic Retries

The behavior of automatic retries is intimately tied to the presence (and absence) of connections. This section describes the errors that cause Ice to close connections, and provides more details about how connections influence retries.

Connection Errors

Ice automatically closes a connection in response to certain fatal error conditions. Of these, the one that is the most likely to affect Ice applications is a connection timeout. Other errors that prompt Ice to close a connection include the following:

- a socket failure while performing I/O on the connection
- receiving an improperly formatted message
- dispatching an operation to a Java servant raises `OutOfMemoryError` or `AssertionError`

When Ice closes a connection in response to one of these errors, all other outstanding requests on the same connection also fail and may be retried if eligible.

Back to Top ^

Connection Status

One factor that influences retry behavior is the status of the connection on which the failed request was attempted. If the failure caused Ice to close the connection (as discussed in the previous section), or if the request failed because Ice could not establish a connection, Ice must try to obtain another connection before it can retry the request.

It is also important to understand that Ice may not retry the invocation on the original endpoint *even if the connection that was used for the initial request remains open*. The retry behavior in this case depends on several criteria:

- whether the proxy caches its connection
- whether the proxy contains multiple endpoints
- whether other connections exist to any of the proxy's endpoints
- the proxy's configured endpoint selection type

Generally speaking, you must configure your application carefully if you need fine-grained control over Ice's retry behavior.

Automatic Retries: Direct Proxy versus Indirect Proxies

With a direct proxy, Ice tries to establish a connection using each suitable endpoint of the proxy, and, if this fails, Ice retries these connection attempts (Ice retries once immediately with the default retry configuration).

With an indirect proxy, the retry algorithm is a little bit different:

- Ice first attempts to establish a connection using the endpoints found in its locator cache, with one attempt for each suitable endpoint.
- if this fails, Ice refreshes its locator cache and tries to establish a connection using the refreshed endpoints (this new attempt with just-refreshed endpoints does not count as a retry).
- if all these attempts still fail, Ice refreshes its locator cache again and tries to establish a connection to the re-refreshed endpoints, which represents retry attempt number 1

Back to Top ^

See Also

- Terminology
- Operations
- Invocation Timeouts
- Connection Timeouts
- Connection Establishment
- Getting Started with Glacier2