Advanced Topics for Deprecated AMI Mapping

On this page:

AMI Concurrency Issues

Support for asynchronous invocations in Ice is enabled by the client thread pool, whose threads are primarily responsible for processing reply messages. It is important to understand the concurrency issues associated with asynchronous invocations:

- A callback object must not be used for multiple simultaneous invocations. An application that needs to aggregate information from multiple replies can create a separate object to which the callback objects delegate.
- Calls to the callback object are always made by threads from an Ice thread pool, therefore synchronization may be necessary if the application might interact with the callback object at the same time as the reply arrives. Furthermore, since the Ice run time never invokes callback methods from the client's calling thread, the client can safely make AMI invocations while holding a lock without risk of a deadlock.
- The number of threads in the client thread pool determines the maximum number of simultaneous callbacks possible for asynchronous invocations. The default size of the client thread pool is one, meaning invocations on callback objects are serialized. If the size of the thread pool is increased, the application may require synchronization, and replies can be dispatched out of order. The client thread pool can also be configured to serialize messages received over a connection so that AMI replies from a connection are dispatched in the order they are received.
- A limitation of AMI is the lack of support for collocation optimization. As a result, AMI invocations are always sent "over the wire" and thus are dispatched by the server thread pool.

Flow Control using Deprecated AMI Mapping

The Ice run time queues asynchronous requests when necessary to avoid blocking the calling thread, but places no upper limit on the number of queued requests or the amount of memory they can consume. To prevent unbounded memory utilization, Ice provides the infrastructure necessary for an application to implement its own flow-control logic by combining the following API components:

- The return value of the asynchronous proxy method
- The `ice_sent` method in the AMI callback object

The return value of the proxy method determines whether the request was queued. If the proxy method returns true, no flow control is necessary because the request was accepted by the local transport buffer and therefore the Ice run time did not need to queue it. In this situation, the Ice run time does not invoke the `ice_sent` method on the callback object; the return value of the proxy method is sufficient notification that the request was sent.

If the proxy method returns false, the Ice run time has queued the request. Now the application must decide how to proceed with subsequent invocations:

- The application can be structured so that at most one request is queued. For example, the next invocation can be initiated when the `ice_sent` method is called for the previous invocation.
- A more sophisticated solution is to establish a maximum allowable number of queued requests and maintain a counter (with appropriate synchronization) to regulate the flow of invocations.

Naturally, the requirements of the application must dictate an implementation strategy.

Implementing `ice_sent` in C++

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object must also derive from the C++ class `Ice::AMISentCallback`:

C++

```
namespace Ice {
    class AMISentCallback {
    public:
        virtual ~AMISentCallback();
        virtual void ice_sent() = 0;
    };
}
```

We can modify our sample client to include an `ice_sent` callback as shown below:

C++

```
class AMI_Model_interpolateI : public Demo::AMI_Model_interpolate,
                               public Ice::AMISentCallback
{
public:
    // ...

    virtual void ice_sent()
    {
        cout << "request sent successfully" << endl;
    }
};
```

Implementing `ice_sent` in Java

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object must also implement the Java interface `Ice.AMISentCallback`:

Java

```
package Ice;

public interface AMISentCallback {
    void ice_sent();
}
```

We can modify our sample client to include an `ice_sent` callback as shown below:

Java

```
class AMI_Model_interpolateI extends Demo.AMI_Model_interpolate
                             implements Ice.AMISentCallback {
    // ...

    public void ice_sent()
    {
        System.out.println("request sent successfully");
    }
}
```

Implementing `ice_sent` in C#

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object must also implement the C# interface `Ice.AMISentCallback`:

C#

```
namespace Ice {
    public interface AMISentCallback
    {
        void ice_sent();
    }
}
```

We can modify our sample client to include an `ice_sent` callback as shown below:

C#

```
class AMI_Model_interpolateI : Demo.AMI_Model_interpolate,
                               Ice.AMISentCallback {
    // ...

    public void ice_sent()
    {
        Console.Out.WriteLine("request sent successfully");
    }
}
```

Implementing `ice_sent` in Python

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object need only define the `ice_sent` method.

We can modify our sample client to include an `ice_sent` callback as shown below:

Python

```
class AMI_Model_interpolateI(object):
    # ...

  def ice_sent(self):
      print "request sent successfully"
```

Flushing Batch Requests using Deprecated AMI Mapping

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides an asynchronous version of this method for applications that wish to flush batch requests without the risk of blocking.

The proxy method `ice_flushBatchRequests_async` initiates an asynchronous flush. Its only argument is a callback object; this object must define an `ice_exception` method for receiving a notification if an error occurs before the message is sent.

If the application is interested in flow control, the return value of `ice_flushBatchRequests_async` is a boolean indicating whether the message was sent synchronously. Furthermore, the callback object can define an `ice_sent` method that is invoked when an asynchronous flush completes.

C++ Mapping

The base proxy class `ObjectPrx` defines the asynchronous flush operation as shown below:

C++

```
namespace Ice {
    class ObjectPrx : ... {
    public:
        // ...
        bool ice_flushBatchRequests_async(
            const Ice::AMI_Object_ice_flushBatchRequestsPtr& cb)
    };
}
```

The argument is a smart pointer for an object that implements the following class:

C++

```
namespace Ice {
    class AMI_Object_ice_flushBatchRequests : ... {
    public:
        virtual void ice_exception(const Ice::Exception& ex) = 0;
    };
}
```

As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

C++

```
class MyFlushCallbackI : public Ice::AMI_Object_ice_flushBatchRequests,
                         public Ice::AMISentCallback
{
public:
    virtual void ice_exception(const Ice::Exception& ex);
    virtual void ice_sent();
};
```

Java Mapping

The base proxy class `ObjectPrx` defines the asynchronous flush operation as shown below:

Java

```
package Ice;

public class ObjectPrx ... {
    // ...
    boolean ice_flushBatchRequests_async(AMI_Object_ice_flushBatchRequests cb);
}
```

The argument is a reference for an object that implements the following class:

Java

```java
package Ice;

public abstract class AMI_Object_ice_flushBatchRequests ...
{
    public abstract void ice_exception(LocalException ex);
}
```

As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

Java

```java
class MyFlushCallbackI extends Ice.AMI_Object_ice_flushBatchRequests
                       implements Ice.AMISentCallback
{
    public void ice_exception(Ice.LocalException ex) { ... }
    public void ice_sent() { ... }
}
```

C# Mapping

The base proxy class `ObjectPrx` defines the asynchronous flush operation as shown below:

C#

```csharp
namespace Ice {
    public class ObjectPrx : ... {
        // ...
        bool ice_flushBatchRequests_async(AMI_Object_ice_flushBatchRequests cb);
    }
}
```

The argument is a reference for an object that implements the following class:

C#

```csharp
namespace Ice {
    public abstract class AMI_Object_ice_flushBatchRequests ... {
        public abstract void ice_exception(Ice.Exception ex);
    }
}
```

As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

C#

```csharp
class MyFlushCallbackI : Ice.AMI_Object_ice_flushBatchRequests,
                         Ice.AMISentCallback
{
    public override void
    ice_exception(Ice.LocalException ex) { ... }

    public void ice_sent() { ... }
}
```

Python Mapping

The base proxy class defines the asynchronous flush operation as shown below:

Python

```
def ice_flushBatchRequests_async(self, cb)
```

The `cb` argument represents a callback object that must implement an `ice_exception` method. As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

Python

```
class MyFlushCallbackI(object):
    def ice_exception(self, ex):
        # handle an exception

    def ice_sent(self):
        # flush has completed
```

Handling Timeouts with Deprecated AMI Mapping

Timeouts for asynchronous invocations behave like those for synchronous invocations: an `Ice::TimeoutException` is raised if the response is not received within the given time period. In the case of an asynchronous invocation, however, the exception is reported to the `ice_exception` method of the invocation's callback object. For example, we can handle this exception in C++ as shown below:

C++

```
class AMI_Model_interpolateI : public Demo::AMI_Model_interpolate
{
public:
    // ...

    virtual void ice_exception(const Ice::Exception& ex)
    {
        try {
            ex.ice_throw();
        } catch (const Demo::RangeError& e) {
            cerr << "interpolate failed: range error" << endl;
        } catch (const Ice::TimeoutException&) {
            cerr << "interpolate failed: timeout" << endl;
        } catch (const Ice::LocalException& e) {
            cerr << "interpolate failed: " << e << endl;
        }
    }
};
```

Handling Errors with Deprecated AMI Mapping

It is important to remember that all errors encountered by an AMI invocation (except `CommunicatorDestroyedException`) are reported back via the `ice_exception` callback, even if the error condition is encountered "on the way out", when the operation is invoked. The reason for this is consistency: if an invocation, such as `foo_async` could throw exceptions, you would have to handle exceptions in two places in your code: at the point of call for exceptions that are encountered "on the way out", and in `ice_exception` for error conditions that are detected after the call is initiated.

Where this matters is if you want to send off a number of AMI calls, each of which depends on the preceding call to have succeeded. For example:

C++

```
p1->foo_async(cb1);
p2->bar_async(cb2);
```

If `bar` depends for its correct working on the successful completion of `foo`, this code will not work because the `bar` invocation will be sent regardless of whether `foo` failed or not.

In such cases, where you need to be sure that one call is dispatched only if a preceding call succeeds, you must instead invoke `bar` from within `foo`'s `ice_response` implementation, instead of from the main-line code.

AMI Limitations

AMI invocations cannot be sent using collocated optimization. If you attempt to invoke an AMI operation using a proxy that is configured to use collocation optimization, the Ice run time will raise `CollocationOptimizationException` if the servant happens to be collocated; the request is sent normally if the servant is not collocated. This optimization is enabled by default (as specified by the `Ice.Default.CollocationOptimized` property) but can be disabled on individual proxies using a proxy method.

See Also

- Location Transparency
- The Ice Threading Model
- Thread Pool Design Considerations
- Deprecated AMI Language Mappings