Asynchronous Method Invocation (AMI) in Java

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

As of version 3.4, Ice provides a new API for asynchronous method invocation. This page describes the new API. Note that the old API is deprecated and will be removed in a future release.

On this page:

Basic Asynchronous API in Java

Consider the following simple Slice definition:

Slice

```
module Demo {
    interface Employees {
        string getName(int number);
    };
};
```

Asynchronous Proxy Methods in Java

Besides the synchronous proxy methods, `slice2java` generates the following asynchronous proxy methods:

Java

```
public interface EmployeesPrx extends Ice.ObjectPrx
{
    // ...

    public Ice.AsyncResult begin_getName(int number);
    public Ice.AsyncResult begin_getName(int number, java.util.Map<String, String> __ctx);

    public String end_getName(Ice.AsyncResult __result);
}
```

Four additional overloads of `begin_getName` are generated for use with generic completion callbacks and type-safe completion callbacks.

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. (The `begin_` method is overloaded so you can pass a per-invocation context.)

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

Java

```
EmployeesPrx e = ...;
Ice.AsyncResult r = e.begin_getName(99);

// Continue to do other things here...

String name = e.end_getName(r);
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `AsyncResult`. This value contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResult` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. Similarly, the `end_` method has one out-parameter for each out-parameter of the corresponding Slice operation (plus the `AsyncResult` parameter). For example, consider the following operation:

Slice

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

Java

```
Ice.AsyncResult begin_op(int inp1, String inp2);
Ice.AsyncResult begin_op(int inp1, String inp2, java.util.Map<String, String> __ctx);
double end_op(Ice.BooleanHolder outp1, Ice.LongHolder outp2, Ice.AsyncResult r);
```

Asynchronous Exception Semantics in Java

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `java.lang.IllegalArgumentException`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

`AsyncResult` Class in Java

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

Java

```
public class AsyncResult {
    public Communicator getCommunicator();
    public Connection getConnection();
    public ObjectPrx getProxy();
    public String getOperation();

    public boolean isCompleted();
    public void waitForCompleted();

    public boolean isSent();
    public void waitForSent();

    public void throwLocalException();

    public boolean sentSynchronously();
}
```

The methods have the following semantics:

- `Communicator getCommunicator()`
  This method returns the communicator that sent the invocation.
- `Connection getConnection()`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `getConnection` method only returns a non-nil value when the `AsyncResult` object is obtained by calling `begin_flushBatchRequests` on a `Connection` object.
- `ObjectPrx getProxy()`
  This method returns the proxy that was used to call the `begin_` method, or nil if the `AsyncResult` object was not obtained via an asynchronous proxy invocation.
- `String getOperation()`
  This method returns the name of the operation.
- `boolean isCompleted()`
  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.
- `void waitForCompleted()`
  This method blocks the caller until the result of an invocation becomes available.
- `boolean isSent()`
  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `isSent` returns false.
- `void waitForSent()`
  This method blocks the calling thread until a request has been written to the client-side transport, or an exception occurs. After `waitForSent` returns, `isSent` returns true if the request was successfully written to the client-side transport, or false if an exception occurred. In the case of a failure, you can call the corresponding `end_` method or `throwLocalException` to obtain the exception.
- `void throwLocalException()`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.
- `boolean sentSynchronously()`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

Polling for Completion in Java

The `AsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
};
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

Java

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;
while (!file.eof()) {
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

Java

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;

LinkedList<Ice.AsyncResult> results = new LinkedList<Ice.AsyncResult>();
int numRequests = 5;

while (!file.eof()) {
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice.AsyncResult r = ft.begin_send(offset, bs);
    offset += bs.length;

    // Wait until this request has been passed to the transport.
    r.waitForSent();
    results.add(r);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while (results.size() > numRequests) {
        Ice.AsyncResult r = results.getFirst();
        results.removeFirst();
        r.waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while (results.size() > 0) {
    Ice.AsyncResult r = results.getFirst();
    results.removeFirst();
    r.waitForCompleted();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Generic Completion Callbacks in Java

The `begin_` method is overloaded to allow you to provide completion callbacks. Here are the corresponding methods for the `getName` operation:

Java

```
Ice.AsyncResult begin_getName(int number, Ice.Callback __cb);

Ice.AsyncResult begin_getName(int number,
                              java.util.Map<String, String> __ctx,
                              Ice.Callback __cb);
```

The second version of `begin_getName` lets you override the default context. Following the in-parameters, the `begin_` method accepts a parameter of type `Ice.Callback`, which is a callback class with a `completed` method that you must provide. The Ice run time invokes the `completed` method when an asynchronous operation completes. For example:

Java

```
public class MyCallback extends Ice.Callback
{
    public void completed(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try {
            String name = e.end_getName(r);
            System.out.println("Name is: " + name);
        } catch (Ice.LocalException ex) {
            System.err.println("Exception is: " + ex);
        }
    }
}
```

Note that your callback class must derive from `Ice.Callback`. The implementation of your callback method must call the `end_` method. The proxy for the call is available via the `getProxy` method on the `AsyncResult` that is passed by the Ice run time. The return type of `getProxy` is `Ice.ObjectPrx`, so you must down-cast the proxy to its correct type.

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. If an operation can throw user exceptions, this means that you need an additional catch handler for `Ice.UserException` (or catch all possible user exceptions explicitly). If you allow an exception to escape from the callback method, the Ice run time produces a log entry by default and ignores the exception. (You can disable the log message by setting the property `Ice.Warn.AMICallback` to zero.)

To inform the Ice run time that you want to receive a callback for the completion of the asynchronous call, you pass the callback instance to the `begin_` method:

Java

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb);
```

This is often written using an anonymous class instead:

Java

```
EmployeesPrx e = ...;

e.begin_getName(
        99,
        new Ice.AsyncCallback()
        {
            public void completed(Ice.AsyncResult r)
            {
                EmployeesPrx p = (EmployeesPrx)r.getProxy();
                try {
                    String name = p.end_getName(r);
                    System.out.println("Name is: " + name);
                } catch (Ice.LocalException ex) {
                    System.err.println("Exception: " + ex);
                }
            }
        });
```

An anonymous class is useful particularly for callbacks that do only a small amount of work because the code that starts the call and the code that processes the results are physically close together.

Sharing State Between `begin_` and `end_` Methods in Java

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

Assuming that we have a `Widget` class that designates a particular user interface element, you could pass different widgets by storing the widget to be used as a member of your callback class:

Java

```
public class MyCallback extends Ice.AsyncCallback
{
    public MyCallback(Widget w)
    {
        _w = w;
    }

    private Widget _w;

    public void completed(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try {
            String name = e.end_getName(r);
            _w.writeString(name);
        } catch (Ice.LocalException ex) {
            System.err.println("Exception is: " + ex);
        }
    }
}
```

For this example, we assume that widgets have a `writeString` method that updates the relevant UI element.

When you call the `begin_` method, you pass the appropriate callback instance to inform the `end_` method how to update the display:

Java

```
EmployeesPrx e = ...;
Widget widget1 = ...;
Widget widget2 = ...;

// Invoke the getName operation with different widget callbacks.
e.begin_getName(99, new MyCallback(widget1));
e.begin_getName(24, new MyCallback(widget2));
```

The callback class provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same callback instance to multiple invocations. (If you do this, your callback methods may need to use synchronization.)

Type-Safe Completion Callbacks in Java

The generic callback API is not entirely type-safe:

- You must down-cast the return value of `getProxy` to the correct proxy type before you can call the `end_` method.

- You must call the correct `end_` method to match the operation called by the `begin_` method.
- You must remember to catch exceptions when you call the `end_` method; if you forget to do this, you will not know that the operation failed.

`slice2java` generates an additional type-safe API that takes care of these chores for you. To use type-safe callbacks, you must implement a callback class that provides two callback methods:

- a `response` method that is called if the operation succeeds
- an `exception` method that is called if the operation raises an exception

Your callback class must derive from the base class that is generated by `slice2java`. The name of this base class is `<module>.Callback_<interface>_<operation>`. Here is a callback class for an invocation of the `getName` operation:

Java

```
public class MyCallback extends Demo.Callback_Employees_getName
{
    public void response(String name)
    {
        System.out.println("Name is: " + name);
    }

    public void exception(Ice.LocalException ex)
    {
        System.err.println("Exception is: " + ex);
    }
}
```

The `response` callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the `respons` callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

The `exception` callback is invoked if the invocation fails because of an Ice run time exception. If the Slice operation can also raise user exceptions, your callback class must supply an additional overloading of `exception` that accepts an argument of type `Ice.UserException`.

The proxy methods are overloaded to accept this callback instance:

Java

```
Ice.AsyncResult begin_getName(int number,
                              Callback_Employees_getName __cb);

Ice.AsyncResult begin_getName(int number,
                              java.util.Map<String, String> __ctx,
                              Callback_Employees_getName __cb);
```

You pass the callback to an invocation as you would with the generic API:

Java

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb);
```

Asynchronous Oneway Invocations in Java

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws an `IllegalArgumentException`.

The callback methods looks exactly as for a twoway invocation. For the generic API, the Ice run time does not call the `completed` callback method unless the invocation raised an exception during the `begin_` method ("on the way out"). For the type-safe API, the `response` method is never called.

Flow Control in Java

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

For the generic API, you can override the `sent` method:

Java

```
public class MyCallback extends Ice.AsyncCallback
{
    public void completed(Ice.AsyncResult r)
    {
        // ...
    }

    public void sent(Ice.AsyncResult r)
    {
        // ...
    }
}
```

You inform the Ice run time that you want to be informed when a call has been passed to the local transport as usual:

Java

```
e.begin_getName(99, new MyCallback());
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the `sent` method from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the `sent` method from a different thread once it has written the request to the local transport. In addition, you can find out from the `AsyncResult` that is returned by the `begin_` method whether the request was sent synchronously or was queued, by calling `sentSynchronously`.

For the generic API, the `sent` method has the following signature:

Java

```
void sent(Ice.AsyncResult r);
```

For the type-safe API, the signature is:

Java

```
void sent(boolean sentSynchronously);
```

For the generic API, you can find out whether the request was sent synchronously by calling `sentSynchronously` on the `AsyncResult`. For the type-safe API, the boolean `sentSynchronously` parameter provides the same information.

The `sent` methods allow you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Asynchronous Batch Requests in Java

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 asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency Semantics for AMI in Java

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or parameter.

AMI Limitations in Java

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 raises `CollocationOptimizationException` if the servant happens to be collocated; the request is sent normally if the servant is not collocated. You can disable this optimization if necessary.

See Also

- Request Contexts
- Batched Invocations
- Location Transparency