Asynchronous Method Invocation (AMI) in Python

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 Python

Consider the following simple Slice definition:

Slice

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

Asynchronous Proxy Methods in Python

Besides the synchronous proxy methods, the Python mapping generates the following asynchronous proxy methods:

Python

```
def begin_getName(self, number, _response=None, _ex=None, _sent=None, _ctx=None)
def end_getName(self, result)
```

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. The `begin_ method` optionally accepts a per-invocation context and callbacks.

- 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:

Python

```
e = EmployeePrx.checkedCast(...)
r = e.begin_getName(99)

# Continue to do other things here...

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. The `end_` method accepts the `AsyncResult` object as its only argument and returns the out-parameters using the same semantics as for regular synchronous invocations. 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:

Python

```
def begin_op(self, inp1, inp2, ...)
def end_op(self, result)
```

The call to `end_op` returns the following tuple:

Python

```
doubleValue, outp1, outp2 = p.end_op(result)
```

Asynchronous Exception Semantics in Python

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 `RuntimeError`. 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 Python

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

Python

```
class AsyncResult:
    def getCommunicator()
    def getConnection()
    def getProxy()
    def getOperation()

    def isCompleted()
    def waitForCompleted()

    def isSent()
    def waitForSent()

    def throwLocalException()

    def sentSynchronously()
```

The methods have the following semantics:

- `getCommunicator()`
  This method returns the communicator that sent the invocation.

- `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.

- `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.

- `getOperation()`
  This method returns the name of the operation.

- `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.

- `waitForCompleted()`
  This method blocks the caller until the result of an invocation becomes available.

- `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.

- `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.

- `throwLocalException()`
  This method throws the local exception that caused the invocation to fail. If no exception has occurred yet, `throwLocalException` does nothing.

- `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 Python

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:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0
while not file.eof():
    bytes = file.read(chunkSize)  # Read a chunk
    ft.send(offset, bytes)        # Send the chunk
    offset += len(bytes.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:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0

results = []
numRequests = 5

while not file.eof():
    bytes = file.read(chunkSize) # Read a chunk

    # Send up to numRequests + 1 chunks asynchronously.
    r = ft.begin_send(offset, bytes)
    offset += len(bytes)

    # Wait until this request has been passed to the transport.
    r.waitForSent()
    results.append(r)

    # Once there are more than numRequests, wait for the least
    # recent one to complete.
    while len(results) > numRequests:
        r = results[0]
        del results[0]
        r.waitForCompleted()

# Wait for any remaining requests to complete.
while len(results) > 0:
    r = results[0]
    del results[0]
    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.

Completion Callbacks in Python

The `begin_` method accepts three optional callback arguments that allow you to be notified asynchronously when a request completes. Here are the corresponding methods for the `getName` operation:

Python

```
def begin_getName(self, number, _response=None, _ex=None, _sent=None, _ctx=None)
```

The value you pass for the response callback (`_response`), the exception callback (`_ex`), or the sent callback (`_sent`) argument must be a *callable object* such as a function or method. The response callback is invoked when the request completes successfully, and the exception callback is invoked when the operation raises an exception. (The sent callback is primarily used for flow control.)

For example, consider the following callbacks for an invocation of the `getName` operation:

Python

```
def getNameCB(name):
    print "Name is: " + name

def failureCB(ex):
    print "Exception is: " + str(ex)
```

The response callback parameters depend on the operation signature. If the operation has a non-`void` return type, the first parameter of the response 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, or if the operation raises a user exception.

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

Python

```
e = EmployeesPrx.checkedCast(...)

e.begin_getName(99, getNameCB, failureCB)
```

Although the signature of an asynchronous proxy method implies that all of the callbacks are optional and therefore can be supplied in any combination, Ice enforces the following semantics at run time:

- If you omit all callbacks, you must call the `end_` method explicitly as described earlier.
- If you supply either a response callback or a sent callback (or both), you must also supply an exception callback.
- You may omit the response callback for an operation that returns no data (that is, an operation with a `void` return type and no out-parameters).

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

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 a callback class:

Python

```
class MyCallback(object):
    def __init__(self, w):
        self._w = w

    def getNameCB(self, name):
        self._w.writeString(name)

    def failureCB(self, ex):
        print "Exception is: " + str(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:

Python

```
e = EmployeesPrx.checkedCast(...)
widget1 = ...
widget2 = ...

# Invoke the getName operation with different widget callbacks.
cb1 = MyCallback(widget1)
e.begin_getName(99, cb1.getNameCB, cb1.failureCB)
cb2 = MyCallback(widget2)
e.begin_getName(24, cb2.getNameCB, cb2.failureCB)
```

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.)

For those situations in which a stateless callback is preferred, you can use a lambda function to pass state to a callback. Consider the following example:

Python

```
def getNameCB(name, w):
    w.writeString(name)

def failureCB(ex):
    print "Exception is: " + str(ex)

e = EmployeesPrx.checkedCast(...)
widget1 = ...
widget2 = ...

# Use lambda functions to pass state.
e.begin_getName(99, lambda name: getNameCB(name, widget1), failureCB)
e.begin_getName(24, lambda name: getNameCB(name, widget2), failureCB)
```

This strategy eliminates the need to encapsulate shared state in a callback class. Since lambda functions can refer to variables in the enclosing scope, they provide a convenient way to pass state directly to your callback.

Asynchronous Oneway Invocations in Python

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 a `RuntimeError`.

The callback signatures look exactly as for a twoway invocation, but the response method is never called and may be omitted.

Flow Control in Python

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.

You can supply a sent callback to be notified when the request was successfully sent:

Python

```
def response(name):
    # ...

def exception(ex):
    # ...

def sent(sentSynchronously):
    # ...
```

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

Python

```
e.begin_getName(99, response, exception, sent)
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the sent callback 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 callback from a different thread once it has written the request to the local transport. The boolean `sentSynchronously` parameter indicates whether the request was sent synchronously or was queued.

The sent callback allows 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 Python

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 Python

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` parameter.

See Also

- Python Mapping for Operations
- Request Contexts
- Batched Invocations