A New Asynchronous Method Invocation API for Ice for Python

Michi Henning, Chief Scientist, ZeroC, Inc.

Introduction

Ice has had an API for asynchronous method invocation (AMI) since its inception. However, that API is quite verbose and not all that easy to use, as well as inflexible. With the release of Ice 3.4, ZeroC introduces a new API for asynchronous method invocation that does not suffer from these problems and provides programmers with far more choice as to how they can structure their code. The new API is available for Python, Java, .NET, and C++.

This article provides an overview of the new Python API and explains its most important features. (As always, you should consult the Ice Manual for complete documentation.) Companion articles describe the corresponding new APIs for Java, .NET and C++.

The old API is still available and it is possible to use both the new and the old API in the same program. This allows you to gradually migrate code away from the old API without having to change all your code at once. For new code, we strongly recommend that you use the new API. (The old API is deprecated and will eventually be removed entirely.)

Contents

The Problems of the Old API

Suppose you have the following Slice definition:

```
// Slice
module Demo
{
    ["ami"]
    interface Employees
    {
        string getName(int number);
    };
};
```

With the old API, to allow you to invoke `getName` asynchronously, **`slice2py`**[1] generates the following proxy method:

```
class EmployeesPrx(Ice.ObjectPrx):
    def getName_async(self, _cb, number, _ctx=None)
```

Note that the `getName_async` method accepts an optional per-invocation context that overrides the default context. (The new API also permits you to supply a context, but we do not discuss contexts further in this article. Please see the Ice Manual for a full description.)

To call `getName` asynchronously, you must implement a class that implements two methods, `ice_response` and `ice_exception`. These methods are the callback methods that are called by the Ice run time to inform you when an asynchronous call of `getName` completes. For example:

```
class AMI_Employees_getNameI(object):
    def ice_response(self, name):
        print "Name is: " + name

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

When you call `getName` asynchronously, you must supply an instance of this callback class, for example:

```
e = EmployeesPrx.checkedCast(...)
cb = AMI_Employees_getNameI()
e.getName_async(cb, 99)
```

This invokes the `getName` operation in the server without blocking the caller; some time later, once the call completes, the Ice run time calls `ice_response` on the callback instance (if the call succeeded) or `ice_exception` (if the call failed).

This is fine as far as it goes. However, there are a number of disadvantages with this API:

- You must implement a separate class for every operation you call asynchronously.
- The class must implement two methods with the names `ice_response` and `ice_exception`.

[1] In this article we refer to static translation using **`slice2py`**, but the Python mapping is the same regardless of whether the code is generated statically by **`slice2py`** or generated dynamically at run time.

- There is no way to poll for completion of an asynchronous call.
- There is no simple way to block until a particular call completes. (To achieve this, you have to use a monitor that is signaled from the `ice_response` and `ice_exception` callbacks.)

The above points boil down to two major points: verbosity and inflexibility. The verbosity is not immediately obvious, but becomes clear when you consider an application of more realistic size. For example, if you have eight interfaces with six operations each (all of which you want to call asynchronously), you must implement 48 classes and 96 methods. This is tedious, to say the least.

The inflexibility of the API also is a concern: you cannot poll for call completion and you cannot easily block until a particular call is complete. The *only* way to get the results of an invocation is an asynchronous callback. Moreover, there is no easy way to share code on a call-by-call basis. For example, if in a particular section of your application, you want to treat a particular error condition in a different way, you must write a new callback class for every operation that implements the different error handling (or, alternatively, pass additional state into your callback instance so it can select inside the `ice_exception` callback which behavior is needed).

In summary, the old API is as rigid as steel: there is one and only one way of doing things. This is inconvenient if, for example, you have many simple get/set operations that substantially perform the same actions when they complete, or if you would like to block until a particular invocation or group of invocations is complete.

The New Approach

The new asynchronous API eliminates both the verbosity and inflexibility of the old API. Not only can you do more things with the new API, but you can also do so with less code. In turn, this reduces development time and maintenance effort.

The following sections provide an overview of the capabilities of the new API and show some examples of how and why you might want to use a particular API feature.

No Need for Metadata

Here is our simple `Employees` interface once more:

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

Note that the `["ami"]` metadata directive is absent. This is because the new asynchronous API is always generated by **`slice2py`**, so no separate metadata directive is necessary. (If you do add the `["ami"]` directive, **`slice2py`** generates both the old and the new API; you can use both APIs in the same program.)

Basic Asynchronous Invocations

The asynchronous proxy methods look as follows:

```
class EmployeesPrx(Ice.ObjectPrx):
    def begin_getName(self, number, _response=None,
                      _ex=None, _sent=None, _ctx=None)
    def end_getName(self, _result)
```

Note that the `getName` operation now has a `begin_` method and an `end_` method. The `begin_getName` method starts the asynchronous call and is guaranteed not to block the caller. The `end_getName` method is used to collect the result of the invocation. If, at the time the client calls `end_getName`, the operation is not complete yet, `end_getName` blocks the calling thread until the call is complete. On the other hand, if the call completed earlier, some time after the client called `begin_getName` but before it calls `end_getName`, `end_getName` completes immediately.

The `begin_` method has several optional parameters: callback methods (which we will discuss shortly) and a context (see the Ice Manual for details).

Here is an example of how a client can make such an asynchronous invocation:

```
e = EmployeesPrx.checkedCast(...)
result = e.begin_getName(99) # Does not block

# Continue to do other things here...

name = e.end_getName(result) # Blocks until result is available
```

Now, at first glance, this does not look particularly useful: if you call `end_getName` immediately after calling `begin_getName`, you get the same effect as having used an ordinary synchronous call.

However, because `begin_getName` is guaranteed not to block, you can continue to do other things. This is useful if you know that a particular operation invocation may take some time, and there are other tasks you can perform before you need to collect the result of the invocation. That way, you get increased concurrency between client and server without having to use separate threads.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. The `end_` method accepts as its only argument the asynchronous result object, and returns all of the operation's out-parameters as the return value, just as for a regular synchronous invocation.

If an operation raises a user exception or Ice run-time exception, the exception is thrown by the `end_` method. (The `begin_` method does not throw Ice exceptions other than `CommunicatorDestroyedException`.)

Polling and Waiting for Call Completion

Note that the `begin_` method returns an `AsyncResult` instance. This instance encapsulates the state of the asynchronous call and allows you to learn something about the details of the call. You must pass the `AsyncResult` you obtained from the `begin_` method to the corresponding `end_` method. The information in the `AsyncResult` allows the Ice run time to locate the reply from the server and to call the appropriate unmarshaling code. (The unmarshaling of the reply is done by the `end_` method, that is, the Ice run time stores the reply from the server until the `end_` method is called, and the decoding of the reply is done by the `end_` method.)

The `AsyncResult` class contains a few methods that allow you to check call progress and completion:

```
class AsyncResult:
    def sentSynchronously()

    def isSent()
    def waitForSent()

    def isCompleted()
    def waitForCompleted()

    # ...
```

`sentSynchronously` reports whether the Ice run time was able to immediately pass the invocation to the client's local transport. The method returns true if the invocation was written to the local transport immediately; otherwise, the method returns false, indicating that the Ice run time queued the invocation for later transmission because the local transport could not accept it at the time the `begin_` method was called.

`isSent` reports whether, at that time, the invocation has been passed to the client's local transport (whether it was initially queued for transmission or not). `waitForSent` blocks the calling thread until the local transport has accepted the invocation and returns immediately if the invocation was written to the local transport earlier.

The `isCompleted` method returns true if the Ice run time has received the server's reply for the invocation (whether successful or indicating an exception) and false, otherwise. The `waitForCompleted` method blocks the calling thread until the reply from the server has been received and returns immediately if the reply was received earlier.

One way to use these methods is to achieve better concurrency between client and server for data transfers. For example, suppose we have an interface that permits the client to send a file to the server. Because files are often larger than what can be transmitted with a single remote procedure call, the interface allows the client to send the file in chunks:

```
// Slice
module Demo
{
    interface FileTransfer
    {
        void send(int offset, ByteSeq bytes);
    };
};
```

The client can repeatedly call 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:

```
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)
```

This works, but not very well: because the client makes a synchronous call, 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:

```
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 those 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` depends 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

The `begin_` method accepts a number of optional callback arguments that allow you to be notified asynchronously when a request completes. Here is the signature of `begin_getName` once more:

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

The value you pass for the `_response`, `_ex`, or `_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 `_ex` callback is invoked when the operation raises an exception. (We discuss the `_sent` callback shortly.)

For example:

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

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

Note that the return value of the operation becomes an in-parameter of the `_response` callback. (If an operation has out-parameters, these become additional in-parameters to the `_response` callback.)

This style of callback is similar to the old API with its `ice_response` and `ice_exception` methods. However, it is considerably more flexible:

- Your callbacks can be regular Python functions or methods of a class.
- The names of the callbacks can be anything you choose instead of being fixed as `ice_response` and `ice_exception`.
- You can have a callback class that processes the results for different operations instead of having to implement a separate callback class for each operation.
- Common exception handling logic can be shared among a group of operations by using the same failure callback. You can also create more than one failure callback and choose which one will be called if an operation raises an exception when you call the `begin_` method. This makes it easy to select a different error handling strategy depending on where in your code you call an operation.

At the calling end, you pass the callbacks to the `begin_` method:

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

When you supply callbacks to `begin_` as shown above, it is no longer necessary to call the corresponding `end_` method (in fact, doing so raises an exception).

Passing State from the `begin_` Method to the `end_` Method

It is common for applications to have some shared state that is established when an asynchronous call is started, and needed again when that call completes. 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 code calling the `begin_` method knows which user interface element should receive the update, and the code inside `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:

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

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)

# 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)
```

This example assumes that widgets have a `writeString` method that updates the relevant UI element. 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.) This allows you to tailor your classes to match the semantics of your application.

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:

```
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(9, lambda name: getNameCB(name, widget1), failureCB)
e.begin_getName(2, lambda name: getNameCB(name, widget2), failureCB)
```

With this strategy, it is no longer necessary 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.

Flow Control

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.

To receive this notification, you can supply a `_sent` callback:

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

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 by passing the callback instance as usual:

```
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` 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`. The same boolean value is also passed as an argument to the `sent` method.

The `sent` method 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. (See the Ice Manual for more detail.)

Oneway Invocations

The new API permits you to invoke operations via oneway proxies just like you would invoke them via twoway proxies. Because oneway operations do not return a result from the server, the API does not use a response callback (only an exception callback, in case the operation raised an exception "on the way out", that is, in the client-side run time).

Asynchronous Invocation of Operations on `Ice.Object`

The old API did not permit you to invoke remote operations on `Ice.Object` (such as `ice_ping`) asynchronously. The new API rectifies this and permits you to invoke these operations asynchronously.

Concurrency

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. For the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` parameter, so you can take appropriate action to avoid a deadlock.

Summary

The new asynchronous API for Ice is a big improvement over its earlier incarnation. Not only does it provide more features, but it also is less verbose and, due to its flexibility, makes it easier to match the style of interaction with the Ice run time to the needs of your application.

ZeroC will continue to provide the old API for some time. However, be aware that, as of Ice 3.4, the old API is deprecated and that it will eventually be removed. You should therefore use the new API when developing new code, and migrate old code to use the new API.