A New Asynchronous Method Invocation API for Ice for C++

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 C++, Java, .NET, and Python.

This article provides an overview of the new C++ 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 Python.

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, **`slice2cpp`** generates the following proxy member functions:

```
class Employees : virtual public IceProxy::Ice::Object
{
    bool getName_async(const Demo::AMI_Employees_getNamePtr&,
                       Ice::Int);
    bool getName_async(const Demo::AMI_Employees_getNamePtr&,
                       Ice::Int,
                       const Ice::Context&);
    // ...
};
```

Note that the `getName_async` member function is overloaded so you can add a per-invocation context instead of sending 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 derives from the `Demo::AMI_Employees_getName` class that is generated by **`slice2cpp`**. Your derived class must implement two member functions, `ice_response` and `ice_exception`. These member functions 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 : public Demo::AMI_Employees_getName
{
public:
    virtual void ice_response(const std::string& name)
    {
        cout << "Name is: " << name << endl;
    }

    virtual void ice_exception(const Ice::Exception& ex)
    {
        cerr << "Exception is: " << ex << endl;
    }
};
```

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

```
EmployeesPrx e = ...;
AMI_Employees_getNamePtr cb = new 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 member functions with the names `ice_response` and `ice_exception`.
- 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 base class from which you derive your callback class stores the state of the asynchronous call. This means that you cannot use the same callback instance for multiple concurrent calls. (You can reuse a callback instance, but only once the previous invocation using that instance is complete.)

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 member functions. 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. Moreover, the new API reduces the amount of object code that is generated because its implementation makes extensive use of templates. This means that much of the object code for the API is generated only for operations that you actually call asynchronously, instead of being present always, whether that code is actually called or not. The new API is also better tailored to C++. For example, templates provide type-safety and reduce the amount of code you need to write.

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 **`slice2cpp`**, so no separate metadata directive is necessary. (If you do add the `["ami"]` directive, **`slice2cpp`** 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 Employees : virtual public IceProxy::Ice::Object
{
    Ice::AsyncResultPtr begin_getName(Ice::Int);
    std::string end_getName(const Ice::AsyncResultPtr&);
    // ...
};
```

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 API generated by **`slice2cpp`** has six overloads of the `begin_` method (of which we only show the first one in the preceding example); the other overloads deal with contexts (see the Ice Manual for details), as well as callbacks and cookies, which we will discuss shortly.

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

```
EmployeesPrx e = ...;
Ice::AsyncResultPtr r = e->begin_getName(99); // Does not block

// Continue to do other things here...

string name = e->end_getName(r); // 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. Similarly, the `end_` method has one parameter for each out-parameter of the corresponding Slice operation. (If a Slice operation has a return value, the `end_` method returns that value in the same way that a synchronous invocation would.)

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 a smart pointer to 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
{
public:
    bool sentSynchronously() const;

    bool isSent() const;
    void waitForSent();

    bool isCompleted() const;
    void 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:

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
Ice::Int offset = 0;
while(!file.eof())
{
    ByteSeq bs;
    bs = file.read(chunkSize); // Read a chunk
    ft->send(offset, bs);      // Send the chunk
    offset += bs.size();
}
```

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:

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
Ice::Int offset = 0;

list<Ice::AsyncResultPtr> results;
const int numRequests = 5;

while(!file.eof())
{
    ByteSeq bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice::AsyncResultPtr r = ft->begin_send(offset, bs);
    offset += bs.size();

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

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.size() > numRequests)
    {
        Ice::AsyncResultPtr r = results.front();
        results.pop_front();
        r->waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while(!results.empty())
{
    Ice::AsyncResultPtr r = results.front();
    results.pop_front();
    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 is overloaded to allow you to supply a callback method that is called by the Ice run time when an asynchronous invocation completes. To receive a callback, you must implement a class that derives from IceUtil::Shared, for example:

```
class MyCallback : public IceUtil::Shared
{
public:
    void finished(const Ice::AsyncResultPtr&);
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

The callback method can have any name; it must conform to the above signature, that is, have void return type and accept a single parameter of type const Ice::AsyncResultPtr&.

The implementation of the callback method is expected to call the end_ method for the corresponding invocation. For example, you could write the callback method for an invocation of getName as follows:

```
void
MyCallback::finished(const Ice::AsyncResultPtr& r)
{
    EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
    try
    {
        string name = e->end_getName(r);
        cout << "Name is: " << name << endl;
    }
    catch(const Ice::Exception& ex)
    {
        cerr << "Exception is: " << ex << endl;
    }
}
```

Note the getProxy method on the AsyncResult that is passed to the callback method: it returns the proxy that was used to invoke the begin_ method. The return value of getProxy is of type Ice::ObjectPrx, so we need to down-cast it to an EmployeePrx before we can invoke end_getName. In this example, we know that the proxy cannot have any type other than EmployeePrx, so an unchecked cast is sufficient. (In general, you should always use an unchecked cast here because a checked cast would result in an additional invocation to the server to verify that the proxy is of the expected type.)

Having written the completion callback, the question is how we inform the Ice run time that we want it to call `finished` when a `getName` call completes. We achieve this by passing a callback object to the `begin_` method:

```
EmployeesPrx e = ...;

MyCallbackPtr cb = new MyCallback;
Ice::CallbackPtr d = Ice::newCallback(cb, &MyCallback::finished);

e->begin_getName(99, d);
```

Note that this code calls a helper function in the Ice run time, `Ice::newCallback`. This helper function accepts a smart pointer to your callback instance `cb`, plus a member function pointer to specify the method you want to be called when the invocation completes. In this case, we specify the member function pointer of the callback method we defined earlier, `&MyCallback::finished`.

Some time after calling `begin_getName` in this way, the Ice run time calls `finished` on your callback instance and, in turn, `finished` calls the `end_` method to unmarshal the server's reply.

Note that this scheme is considerably more flexible and simpler than the old asynchronous API:

- Your callback class only needs to derive from `IceUtil::Shared` and does not need to derive from a Slice-generated base class.
- The callback method can have any name you prefer.
- There is only a single callback method instead of two.

To see how this reduces the amount of code you need to write, consider a modified version of our `Employees` interface:

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

This is the same interface as previously, but with an additional `getNumber` operation. We can now create a single callback class that can deal with results from invoking either operation, as well as easily share common exception handling:

```
class MyCallback : public IceUtil::Shared
{
public:
    void getName(const Ice::AsyncResultPtr&);
    void getNumber(const Ice::AsyncResultPtr&);

private:
    void handleException(const Ice::Exception&);
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

The implementation of `getName` is identical to the previous `finished` method, but delegates exception handling to the private `handleException` method:

```
void
MyCallback::getName(const Ice::AsyncResultPtr& r)
{
    EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
    try
    {
        string name = e->end_getName(r);
        cout << "Name is: " << name << endl;
    }
    catch(const Ice::Exception& ex)
    {
        handleException(ex);
    }
}
```

The implementation of `getNumber` is identical, except that it calls `end_getNumber` instead of `end_getName`:

```
void
MyCallback::getNumber(const Ice::AsyncResultPtr& r)
{
    EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
    try
    {
        Ice::Int number = e->end_getNumber(r);
        cout << "Number is: " << number << endl;
    }
    catch(const Ice::Exception& ex)
    {
        handleException(ex);
    }
}
```

The calling code instantiates a callback instance as before, but now uses two different `Ice::Callback` instances, one for each operation:

```
EmployeesPrx e = ...;

MyCallbackPtr cb = new MyCallback;

Ice::CallbackPtr getNameCB =
   Ice::newCallback(cb, &MyCallback::getName);
Ice::CallbackPtr getNumberCB =
   Ice::newCallback(cb, &MyCallback::getNumber);

e->begin_getName(99, getNameCB);          // Call getName.
e->begin_getNumber("Fred", getNumberCB); // Call getNumber.

// Call the operations again for a different employee.
```

```
e->begin_getName(42, getNameCB);
e->begin_getNumber("Joe", getNumberCB);
```

This is considerably terser than the old API:

- Considering an application with eight interfaces with six operations each once more, we can now call all of the operations asynchronously by implementing one class with 49 member functions, instead of having to implement 48 classes and 96 member functions. (Moreover, as we will see in a moment, it is also possible to achieve this with a single class with a single member function.)
- The API is more flexible: you are free to choose whatever names you deem most appropriate for your callback methods, and you are free to choose whether you want to use a single callback instance for different operations or create separate callback instances of different types for each operation (or any mix of these two choices).
- Your callback instance and the `Ice::Callback` instances that determine which of your callback methods will be called by the Ice run time are reusable: you can choose which invocation uses what callback method, and you can make a different choice for individual invocations.

This more flexible callback approach is particularly useful if you have shared state that is relevant to a subset of operations: you can add that state to your callback instance so it is available to all the callback methods for these operations. This makes it easy to partition your callbacks into classes and callback methods in the way that most closely matches your application's needs (instead of that choice being dictated to you, as is the case with the old API).

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 the `end_` method needs access to that element.

The asynchronous method invocation API allows you to pass such state by providing a cookie. A cookie is an instance of a class that you write; the class can contain whatever data you want to pass, as well as any methods you may want to add to manipulate that data.

The only requirement on the cookie class is that it must derive from `Ice::LocalObject`. Here is an example implementation that stores a `WidgetHandle`. (We assume that this class provides whatever methods are needed by the `end_` method to update the display.)

```
class Cookie : public Ice::LocalObject
{
public:
    Cookie(WidgetHandle h) : _h(h) {}
    WidgetHandle getWidget() { return _h; }

private:
    WidgetHandle _h;
};
typedef IceUtil::Handle<Cookie> CookiePtr;
```

A the calling end, the client can pass the cookie as an additional parameter to the begin_ method, for example:

```
// Make cookie for call to getName(99).
CookiePtr cookie1 = new Cookie(widgetHandle1);

// Make cookie for call to getName(42);
CookiePtr cookie2 = new Cookie(widgetHandle2);

// Invoke the getName operation with different cookies.
e->begin_getName(99, getNameCB, cookie1);
e->begin_getName(24, getNameCB, cookie2);
```

The end_ method can retrieve the cookie from the AsyncResult by calling getCookie. For this example, we assume that widgets have a writeString method that updates the relevant UI element:

```
void
MyCallback::getName(const Ice::AsyncResultPtr& r)
{
    EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
    CookiePtr cookie = CookiePtr::dynamicCast(r->getCookie());
    try
    {
        string name = e->end_getName(r);
        cookie->getWidget()->writeString(name);
    }
    catch(const Ice::Exception& ex)
    {
        handleException(ex);
    }
}
```

The cookie 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 cookie instance to multiple invocations. This allows you to tailor your classes to match the semantics of your application.

Using a Single Callback Method for Many Operations

The approaches we have explored so far all require you to implement a separate callback method for each operation. However, you can easily create a generic callback method that can handle the results for many different operations.

The actions taken by the callback depend on the operation whose results are being processed: it is the operation that determines the type of the return value and out-parameters (if any). Suppose we want to use a single callback method to process the result of both the getName and getNumber operations on our Employees interface. One way to achieve this is to retrieve the operation name from the AsyncResult and use it to decide which end_ method to call:

```
void
MyCallback::finished(const Ice::AsyncResultPtr& r)
{
    EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
    try
    {
        string op = r->getOperation();
        if(op == "getName")
        {
            string name = e->end_getName(r);
            cout << "Name is: " << name << endl;
        }
        else
        {
            Ice::Int number = e->end_getNumber(r);
            cout << "Number is: " << number << endl;
        }
    }
    catch(const Ice::Exception& ex)
    {
        cerr << "Exception is: " << ex << endl;
    }
}
```

With this implementation, the calling end can use the same callback method for both operations:

```
MyCallbackPtr cb = new MyCallback;
Ice::CallbackPtr genericCB =
   Ice::newCallback(cb, &MyCallback::finished);

e->begin_getName(99, genericCB);
e->begin_getNumber("Fred", genericCB);
```

Another implementation that achieves the same thing is to store a dictionary in the callback class that maps the operation name to an enumerator. The callback method uses the operation name to efficiently retrieve the enumerator (instead of performing a series of string comparisons) and, with a `switch` statement, to select what `end_` method to call and how to process the operation result.

There are many other variations to this theme. For example, you can store a smart pointer to a base class and invoke a pure virtual function via the pointer. For each operation, you derive a class from the base that implements the virtual function and calls the `end_` method.

Yet another option is to use a cookie that stores a base class smart pointer or member function pointer and to invoke the code that calls the `end_` method by retrieving that pointer from the cookie.

Which implementation technique you end up using depends entirely on your application. The point is that the API is flexible and therefore can adapt to your needs. This is particularly useful if you already have a large body of code and want to call that code when the results of an invocation are ready: you can easily choose an implementation that matches your needs instead of having to use the rigid one-size-fits-all approach of the old API.

Type-Safe Callbacks

One thing you may have noticed with the API we have explored so far is that it 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.
- If you use a cookie, you must down-cast the cookie to the correct type before you can access the data inside the cookie.
- 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. (If a callback method throws an exception, the Ice run time ignores the exception after logging a warning.)

This lack of type-safety is the price we pay for the terse and generic nature of the API. If you do not require the flexibility of the generic API, you can instead use a type-safe API. That API is somewhat less flexible but, in return, does not require you to perform any down-casts, select the correct `end_` method, or catch exceptions.

The type-safe API is implemented as a C++ template that calls a success or failure callback method with a strongly-typed signature, depending on whether the operation succeeded or raised an exception. The type-safe API adds no overhead in terms of code size unless you actually use it. If you do use it, the amount of object code that is generated is no larger than what you would get by using a generic callback method that performs the down-casts and calls the `end_` method.

To use type-safe callbacks, you implement a callback class that has one method that receives the operation result if the operation succeeds, and another method that receives the exception that is raised if the operation fails. For example:

```
class MyCallback : public IceUtil::Shared
{
public:
    void getNameCB(const string&);
    void getNumberCB(Ice::Int);
    void failureCB(const Ice::Exception&);
};
```

Note that the callback methods are now strongly typed: the return value of the operations becomes an in-parameter of the correct type. (If an operation has out-parameters, these become additional in-parameters for the callback method.) If either operation raises an exception, the Ice run time calls the `failureCB` method.

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

- The callback class must derive from `IceUtil::Shared`, but it does not need to derive from any Slice-generated code.
- The names of the callback methods 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 method. 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.
- You can use the same callback instance for multiple concurrent invocations—the callback instance does not store any state that is needed by the Ice run time. (If you do this, your callback methods may need to use synchronization.)

Returning to our `Employees` example, here is the implementation of the two success callbacks and the failure callback:

```
void
MyCallback::getNameCB(const string& name)
{
    cout << "Name is: " << name << endl;
}

void
MyCallback::getNumberCB(Ice::Int number)
{
    cout << "Number is: " << number << endl;
}

void
MyCallback::failureCB(const Ice::Exception& ex)
{
    cerr << "Exception is: " << ex << endl;
}
```

At the calling end, you call the `begin_` method as follows:

```
MyCallbackPtr cb = new MyCallback;

Callback_Employees_getNamePtr getNameCB =
    newCallback_Employees_getName(cb,
                                  &MyCallback::getNameCB,
                                  &MyCallback::failureCB);

Callback_Employees_getNumberPtr getNumberCB =
    newCallback_Employees_getNumber(cb,
                                    &MyCallback::getNumberCB,
                                    &MyCallback::failureCB);

e->begin_getName(99, getNameCB);
e->begin_getNumber("Fred", getNumberCB);
```

Note the smart pointers for the type-safe callbacks. These are smart pointers to a template that takes care of invoking the callback methods. The smart pointer type and its template are generated by **`slice2cpp`**. The name of the callback pointer is

`Callback_<interface>_<operation>Ptr`. Also note the `newCallback_Employees_getName` and `newCallback_Employees_getNumber` helper functions. These create a class instance that encapsulates your callback instance and the success and failure member function pointers; the Ice run time uses this instance to invoke your callback methods when an operation completes.

Like the generic API, the type-safe API supports the use of cookies. However, with the type-safe API, you do not need to down-cast the cookie. Instead, the cookie is passed to the callback methods as a smart pointer to its actual type. (Please consult the Ice Manual for details on how to use cookies with the type-safe API.)

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.

For the generic API, you can create an additional callback method:

```
class MyCallback : public IceUtil::Shared
{
public:
    void finished(const Ice::AsyncResultPtr&);
    void sent(const Ice::AsyncResultPtr&);
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

As with any other callback method, you are free to choose any name you like. For this example, the name of the callback method is `sent`. You inform the Ice run time that you want to be informed when a call has been passed to the local transport by specifying the `sent` method as an additional parameter when you create the `Ice::Callback`:

```
EmployeesPrx e = ...;

MyCallbackPtr cb = new MyCallback;
Ice::CallbackPtr d = Ice::newCallback(cb,
                                      &MyCallback::finished,
                                      &MyCallback::sent);

e->begin_getName(99, d);
```

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:

```
void sent(const Ice::AsyncResult&);
```

For the type-safe API, there are two versions, one without and one with a cookie:

```
void sent(bool sentSynchronously);
void sent(bool sentSynchronously, const <CookiePtr>& cookie);
```

For the version with a cookie, *`<CookiePtr>`* is replaced with the actual type of the cookie smart pointer you passed to the `begin_` method.

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. (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 type-safe version of the API does not use a success callback (only a failure 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. This means that you can safely use a non-recursive mutex in a callback method without the risk of self-deadlock. For the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or 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 far 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.