Asynchronous Method Invocation (AMI) in C++

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 C++

Consider the following simple Slice definition:

Slice

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

Asynchronous Proxy Methods in C++

Besides the synchronous proxy methods, `slice2cpp` generates the following asynchronous proxy methods:

C++

```
Ice::AsyncResultPtr begin_getName(Ice::Int number);
Ice::AsyncResultPtr begin_getName(Ice::Int number, const Ice::Context& __ctx)

std::string end_getName(const Ice::AsyncResultPtr&);
```

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

C++

```
EmployeesPrx e = ...;
Ice::AsyncResultPtr 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 `AsyncResultPtr`. The `AsyncResult` associated with this smart pointer contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResultPtr` 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 `AsyncResultPtr` 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:

C++

```
Ice::AsyncResultPtr begin_op(Ice::Int inp1, const ::std::string& inp2)

Ice::Double end_op(bool& outp1, Ice::Long& outp2, const Ice::AsyncResultPtr&);
```

Asynchronous Exception Semantics in C++

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 `IceUtil::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 C++

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

C++

```
class AsyncResult : virtual public IceUtil::Shared, private IceUtil::noncopyable {
public:
    virtual bool operator==(const AsyncResult&) const;
    virtual bool operator<(const AsyncResult&) const;

    virtual Int getHash() const;

    virtual CommunicatorPtr getCommunicator() const;
    virtual ConnectionPtr getConnection() const;
    virtual ObjectPrx getProxy() const;
    const string& getOperation() const;
    LocalObjectPtr getCookie() const;

    bool isCompleted() const;
    void waitForCompleted();

    bool isSent() const;
    void waitForSent();

    void throwLocalException() const;

    bool sentSynchronously() const;
};
```

The methods have the following semantics:

- `bool operator==(const AsyncResult&) const`
  `bool operator<(const AsyncResult&) const`
  `Int getHash() const`
  These methods allow you to create ordered or hashed collections of pending asynchronous invocations. This is useful, for example, if you can have a number of outstanding requests, and need to pass state between the `begin_` and the `end_` methods. In this case, you can use the returned `AsyncResult` objects as keys into a map that stores the state for each call.

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

- `virtual ConnectionPtr getConnection() const`
  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.

- `virtual ObjectPrx getProxy() const`
  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.

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

- `LocalObjectPtr getCookie() const`
  This method returns the cookie that was passed to the `begin_` method. If you did not pass a cookie to the `begin_` method, the return value is null.

- `bool isCompleted() const`
  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.

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

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:

C++

```
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 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:

C++

```
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 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 C++

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

C++

```
Ice::AsyncResultPtr begin_getName(
                        Ice::Int number,
                        const Ice::CallbackPtr& __del,
                        const Ice::LocalObjectPtr& __cookie = 0);

Ice::AsyncResultPtr begin_getName(
                        Ice::Int number,
                        const Ice::Context& __ctx,
                        const Ice::CallbackPtr& __del,
                        const Ice::LocalObjectPtr& __cookie = 0);
```

The second version of `begin_getName` lets you override the default context. (We discuss the purpose of the `cookie` parameter in the next section.) Following the in-parameters, the `begin_` method accepts a parameter of type `Ice::CallbackPtr`. This is a smart pointer to a callback class that is provided by the Ice run time. This class stores an instance of a callback class that you implement. The Ice run time invokes a method on your callback instance when an asynchronous operation completes. Your callback class must provide a method that returns `void` and accepts a single parameter of type `const AsyncResultPtr&`, for example:

C++

```
class MyCallback : public IceUtil::Shared {
public:
    void 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;
        }
    }
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

Note that your callback class must derive from `IceUtil::Shared`. The callback method can have any name you prefer but its signature must match the preceding example.

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. (You should always use an `uncheckedCast` to do this, otherwise you will send an additional message to the server to verify the proxy type.)

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. 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:

C++

```
EmployeesPrx e = ...;

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

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

Note the call to `Ice::newCallback` in this example. This helper function expects a smart pointer to your callback instance and a member function pointer that specifies your callback method.

Using Cookies for Generic Completion Callbacks in C++

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.

The 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.)

C++

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

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

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

C++

```
// 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:

C++

```
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.

Type-Safe Completion Callbacks in C++

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.
- 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.

`slice2cpp` generates an additional type-safe API that takes care of these chores for you. The type-safe API is provided as a template that works much like the `Ice::Callback` class of the generic API, but requires strongly-typed method signatures.

To use type-safe callbacks, you must implement a callback class that provides two callback methods:

- A success callback that is called if the operation succeeds
- A failure callback that is called if the operation raises an exception

As for the generic API, your callback class must derive from `IceUtil::Shared`. Here is a callback class for an invocation of the `getName` operation:

C++

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

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

The callback methods can have any name you prefer and must have `void` return type. The failure callback always has a single parameter of type `const Ice::Exception&`. The success callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the success 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.

To receive these callbacks, you instantiate your callback instance and specify the methods you have defined before passing a smart pointer to a callback wrapper instance to the `begin_` method:

C++

```
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 how this code creates instances of two smart pointer types generated by `slice2cpp` named `Callback_Employees_getNamePtr` and `Callback_Employees_getNumberPtr`. Each smart pointer points to a template instance that encapsulates your callback instance and two member function pointers for the callback methods. The name of this smart pointer type is formed as follows:

`<`*`module`*`>::Callback_<`*`interface`*`>_<`*`operation`*`>Ptr`

Also note that the code uses helper functions to initialize the smart pointers. The first argument to the helper function is your callback instance, and the two following arguments are the success and failure member function pointers, respectively. The name of this helper function is formed as follows:

`<`*`module`*`>::newCallback_<`*`interface`*`>_<`*`operation`*`>`

It is legal to pass a null pointer as the success or failure callback. For the success callback, this is legal only for operations that have `void` return type and no out-parameters. This is useful if you do not care when the operation completes but want to know if the call failed. If you pass a null exception callback, the Ice run time will ignore any exception that is raised by the invocation.

The type of the success and exception member function pointers is provided as `Response` and `Exception` typedefs by the callback template. For example, you can ignore exceptions for an invocation of `getName` as follows:

C++

```
Callback_Employees_getName::Exception nullException = 0;

MyCallbackPtr cb = new MyCallback;

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

e->begin_getName(99, getNameCB); // Ignores exceptions
```

Using Cookies for Type-Safe Completion Callbacks in C++

The `begin_` method optionally accepts a cookie as a trailing parameter. As for the generic API, you can use the cookie to share state between the `begin_` and `end_` methods. However, with the type-safe API, there is no need to down-cast the cookie. Instead, the cookie parameter that is passed to the `end_` method is strongly typed. Assuming that you have defined a `Cookie` class and `CookiePtr` smart pointer, you can pass a cookie to the `begin_` method as follows:

C++

```
MyCallbackPtr cb = new MyCallback;

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

CookiePtr cookie = new Cookie(widgetHandle);
e->begin_getName(99, getNameCB, cookie);
```

The callback methods of your callback class simply add the cookie parameter:

C++

```
class MyCallback : public IceUtil::Shared
{
public:
    void getNameCB(const string& name, const CookiePtr& cookie) {
        cookie->getWidget()->writeString(name);
    }

    void failureCB(const Ice::Exception& ex, const CookiePtr& cookie) {
        cookie->getWidget()->writeError(ex.what());
    }
};
```

Lambda Completion Callbacks in C++

`slice2cpp` generates overloaded versions of the `begin_` methods that accept lambda functions, eliminating the need to create a separate callback object. The Slice compiler generates the following methods for the `getName` operation:

C++

```
Ice::AsyncResultPtr
begin_getName(Ice::Int number,
              const Function<void (const std::string&)>& response,
              const Function<void (const Ice::Exception&)>& exception = ...,
              const Function<void (bool)>& sent = ...);

Ice::AsyncResultPtr
begin_getName(Ice::Int number, const Ice::Context& ctx,
              const Function<void (const std::string&)>& response,
              const Function<void (const Ice::Exception&)>& exception = ...,
              const Function<void (bool)>& sent = ...);

Ice::AsyncResultPtr
begin_getName(Ice::Int number,
              const Function<void (const Ice::AsyncResultPtr&)>& completed,
              const Function<void (const Ice::AsyncResultPtr&)>& sent = ...);

Ice::AsyncResultPtr
begin_getName(Ice::Int number, const Ice::Context& ctx,
              const Function<void (const Ice::AsyncResultPtr&)>& completed,
              const Function<void (const Ice::AsyncResultPtr&)>& sent = ...);
```

The overloaded versions of `begin_getName` allow you to use the type-safe and generic styles, with and without a context argument. Notice that default values are provided for the `exception` and `sent` arguments, meaning you can omit these arguments if you wish to ignore their events. The first two functions above correspond to the type-safe API and the second two have generic API semantics.

These `begin_` methods are generated by default, but can only be used with a compiler that supports C++11 lambda functions, and with an Ice installation that was built with C++11 features enabled.

We can invoke `begin_getName` as shown below:

C++

```
EmployeesPrx e = ...;

// Type-safe
e->begin_getName(42, [](const std::string& name) { cout << "name = " << name << endl; });

// Generic
e->begin_getName(42, [=](const Ice::AsyncResultPtr& r) {
    cout << "name = " << e->end_getName(r) << endl;
});
```

For the sake of brevity we have omitted the error handling that would normally be necessary.

Asynchronous Oneway Invocations in C++

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 `IceUtil::IllegalArgumentException`.

For the generic API, the callback method looks exactly as for a twoway invocation. However, for oneway invocations, the Ice run time does not call the callback method unless the invocation raised an exception during the `begin_` method ("on the way out").

For the type-safe API, the `newCallback` helper for `void` operations is overloaded so you can omit the success callback. For example, here is how you could call `ice_ping` asynchronously:

C++

```
MyCallbackPtr cb = new MyCallback;

Ice::Callback_Object_ice_pingPtr callback =
    Ice::newCallback_Object_ice_ping(cb, &MyCallback::failureCB);

p->begin_opVoid(callback);
```

Flow Control in C++

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:

C++

```
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`:

C++

```
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:

C++

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

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

C++

```
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.

Asynchronous Batch Requests in C++

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 C++

The Ice run time always invokes your callback methods from a separate thread. This means that you can safely use a non-recursive mutex without risking deadlock. There is one exception to this rule: the run time 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, so you can take appropriate action to avoid a self-deadlock.

AMI Limitations in C++

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

- C++ Mapping for Classes
- Smart Pointers for Classes
- Request Contexts
- Batched Invocations
- Location Transparency