Asynchronous Method Invocation (AMI) in Objective-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.

On this page:

- Basic Asynchronous API in Objective-C
 - Proxy Methods for AMI in Objective-C
 - Exception Handling for AMI in Objective-C
- The ICEAsyncResult Protocol in Objective-C
- Polling for Completion in Objective-C
- Completion Callbacks in Objective-C
- Oneway Invocations in Objective-C
- Flow Control in Objective-C
- Batch Requests in Objective-C
- Concurrency in Objective-C

Basic Asynchronous API in Objective-C

Consider the following simple Slice definition:

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

Proxy Methods for AMI in Objective-C

Besides the synchronous proxy methods, the Objective-C mapping generates the following asynchronous proxy methods:

Objective-C

```
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number;
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
context:(ICEContext *)context;
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
response:(void(^)(NSMutableString*))response_
exception:(void(^)(ICEException*))exception_;
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
context:(ICEContext *)context
response:(void(^)(NSMutableString*))response_
exception:(void(^)(ICEException*))exception_;
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
response:(void(^)(NSMutableString*))response_
exception:(void(^)(ICEException*))exception_
sent:(void(^)(BOOL))sent_;
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
context:(ICEContext *)context
response:(void(^)(NSMutableString*))response_
exception:(void(^)(ICEException*))exception_
sent:(void(^)(BOOL))sent_;
-(NSMutableString *) end_getName:(id<ICEAsyncResult>)result;
```

As you can see, the single getName operation results in several begin_getName methods as well as an end_getName method. The begin_ methods optionally accept a per-invocation context and callbacks.

- The begin_getName methods send (or queue) an invocation of getName. These methods do 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:

```
Objective-C
id<EXEmployeesPrx> e = [EXEmployeesPrx checkedCast:...];
id<ICEAsyncResult> r = [e begin_getName:99]
// Continue to do other things here...
NSString* 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 id<ICEAsyncResult>. This value contains the state that the lce run time requires to keep track of the asynchronous invocation. You must pass the id<ICEAsyncResult> 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 id<ICEAsyncRe sult> 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:

Objective-C

```
-(id<ICEAsyncResult>) begin_op:(ICEInt)inpl inp2:(NSString *)inp2;
-(ICEDouble) end_op:(BOOL*)outpl outp2:(ICELong*)outp2
result:(id<ICEAsyncResult>)result;
```

The call to end_op returns the out-parameters as follows:

Objective-C

```
BOOL outp1;
ICELong outp2;
ICEDouble doubleValue = [p end_op:&outp1 outp2:&outp2 result:result];
```

Exception Handling for AMI in Objective-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 IC ECommunicatorDestroyedException. 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 NSException with the NSInvalidArgumentException name. 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 id<ICEAsyncResult> you pass to the end_ method was obtained by calling the begin_ method for a different operation.

The ICEAsyncResult Protocol in Objective-C

The id<ICEAsyncResult> that is returned by the begin_ method encapsulates the state of the asynchronous invocation:

Objective-C

```
@protocol ICEAsyncResult <NSObject>
-(id<ICECommunicator>) getCommunicator;
-(id<ICEConnection>) getConnection;
-(id<ICEObjectPrx>) getProxy;
-(BOOL) isCompleted;
-(void) waitForCompleted;
-(BOOL) isSent;
-(void) waitForSent;
-(BOOL) sentSynchronously;
-(NSString*) getOperation;
@end
```

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 ICEAsyncResult 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 ICEAsyncResult 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 lce run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the lce 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, isSent returns false.

• waitForSent

This method blocks the calling thread until a request has been written to the client-side transport.

• 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, se ntSynchronously 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 Objective-C

The ICEAsyncResult 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:

Objective-C

```
NSInputStream* stream = ...
id<EXFileTransferPrx> ft = [EXFileTransferPrx checkedCast:...];
int chunkSize = ...;
int offset = 0;
while([stream hasBytesAvailable])
{
    char bytes[chunkSize];
    int l = [stream read:bytes maxLength:sizeof(bytes)];
    if(l > 0)
    {
       [ft send:offset bytes:[ByteSeq dataWithBytes:bytes length:l]];
       offset += l;
    }
}
```

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:

Objective-C

```
NSInputStream* stream = ...
id<EXFileTransferPrx> ft = [EXFileTransferPrx checkedCast:...];
int chunkSize = ...;
int offset = 0;
NSMutableArray* results = [NSMutableArray arrayWithCapacity:5];
int numRequests = 5;
while([stream hasBytesAvailable])
{
    char bytes[chunkSize];
   int l = [stream read:bytes maxLength:sizeof(bytes)];
    if(1 > 0)
    {
        // Send up to numRequests + 1 chunks asynchronously.
        id<ICEAsyncResult> r =
            [ft begin_send:offset bytes:[ByteSeq dataWithBytes:bytes length:1]];
        offset += 1;
        // Wait until this request has been passed to the
        // transport.
        [r waitForSent];
        [results addObject:r];
        // Once there are more than numRequests, wait for the
        // least recent one to complete.
        while([results count] > numRequests)
        {
            r = [results objectAtIndex:0];
            [results removeObjectAtIndex:0];
            [r waitForCompleted];
        }
    }
}
// Wait for any remaining requests to complete.
for(id<ICEAsyncResult> r in results)
{
    [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 Objective-C

The begin_ method accepts three optional callback arguments that allow you to be notified asynchronously when a request completes. Here is the signature of the begin_getName method that we saw earlier:

Objective-C

- -(id<ICEAsyncResult>) begin_getName:(ICEInt)number response:(void(^)(NSMutableString*))response_ exception:(void(^)(ICEException*))exception_;
- -(id<ICEAsyncResult>) begin_getName:(ICEInt)number context:(ICEContext *)context response:(void(^)(NSMutableString*))response_ exception:(void(^)(ICEException*))exception_;
- -(id<ICEAsyncResult>) begin_getName:(ICEInt)number response:(void(^)(NSMutableString*))response_ exception:(void(^)(ICEException*))exception_ sent:(void(^)(BOOL))sent_;

```
-(id<ICEAsyncResult>) begin_getName:(ICEInt)number
    context:(ICEContext *)context
    response:(void(^)(NSMutableString*))response_
    exception:(void(^)(ICEException*))exception_
    sent:(void(^)(BOOL))sent_;
```

The value you pass for the response callback (response), the exception callback (exception), or the sent callback (sent) argument must be an Objective-C block. 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:

Objective-C

```
void(^getNameCB)(NSMutableString*) = ^(NSMutableString* name)
{
    NSLog(@"Name is: %@", name);
};
void(^failureCB)(ICEException*) = ^(ICEException* ex)
{
    NSLog(@"Exception is: %@", [ex description]);
};
```

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 called if the invocation fails because of an Ice run time exception, or if the operation raises a user exception.

To inform the lce run time that you want to receive callbacks for the completion of the asynchronous call, you pass the callbacks to the begin_method:

Objective-C

```
e = [EmployeesPrx checkedCast:...]
```

[e begin_getName:99 response:getNameCB exception:failureCB];

You can also pass the Objective-C blocks directly to the call:

Objective-C

```
[e begin_getName:99
    response: ^(NSMutableString* name)
    {
        NSLog(@"Name is: %@", name);
    }
    exception: ^(ICEException* ex)
```

```
{
    NSLog(@"Exception is: %@", [ex description]);
}];
```

Ice enforces the following semantics at run time regarding which callbacks can be optionally specified with a nil value:

- You must 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 outparameters).

Oneway Invocations in Objective-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 NSException with the NSInvalidArgumentException name.

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

Flow Control in Objective-C

Asynchronous method invocations never block the thread that calls the begin_method: the lce 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, [ICEAsyncResult sentSynchronously] returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the lce run time queues the request internally for later transmission in the background. (In that case, [ICEAsyncResult 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:

Objective-C
<pre>void(^sentCB)(BOOL) = ^(BOOL sentSynchronously) { }</pre>

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

Objective-C

[e begin_getName:99 response:getNameCB exception:failureCB sent:sentCB];

If the lce 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 lce run time passes a request to the local transport.

Batch Requests in Objective-C

Applications that send batched requests can either flush a batch explicitly or allow the lce 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 [ICEAsyncResult getConnection]. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency in Objective-C

The lce 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 sent Synchronously parameter.

See Also

- Request Contexts
- Batched Invocations