Asynchronous Method Invocation (AMI) in C++11

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:

Callback and Future-Based APIs

Each Slice operation is mapped to two `<operation-name>Async` functions on the corresponding proxy class:

- a *future-based* function, that returns a `std::future`; this future object delivers the operation's return value and out parameters
- a *callback-based* function, that take callbacks as `std::function` parameters; this is the full featured and somewhat lower-level function

Consider the following simple Slice definition:

Slice

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

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

C++

```
// Future-based function
// with the P = std::promise (the default), it's equivalent to:
// std::future<std::string> getNameAsync(int number, const Ice::Context& context = Ice::noExplicitContext);
//
template<template<typename> class P = std::promise>
auto getNameAsync(int number, const Ice::Context& context = Ice::noExplicitContext)
        -> decltype(std::declval<P<std::string>>().get_future());

// Callback-based function
//
std::function<void()>
getNameAsync(int number,
             std::function<void(std::string)> response,
             std::function<void(std::exception_ptr)> exception = nullptr,
             std::function<void(bool)> sent = nullptr,
             const Ice::Context& context = Ice::noExplicitContext);
```

Future-Based Async Function

The future-based async function returns a `std::future` object. It can also return a custom future object if you specify the associated promise template. For example:

C++

```
auto e = ... // get an Employees proxy
auto fut1 = e->getNameAsync(99); // fut1 is a plain std::future, created from a std::promise
cout << "Employee name is: " << fut1.get() << endl; // fut1.get() blocks until the result is available

auto fut2 = e->getNameAsync<std::experimental::promise>(98); // fut2 is a std::experimental::future,
created from the provided promise template
```

The future's result depends on the operation's parameters:

- when the operation has no return value or out parameter, the result type is `void`.
- when the operation has a return value, or no return value but a single out parameter, the result is this return value or out parameter.
- when the operation has a return value and one or more out parameters (or no return value and two or more out parameters), the result is a generated struct `<`*`operation-name`*`>`*`Result`* (with the first letter capitalized) in the mapped interface class (or in the main mapped class for a class with operations). This struct has public data members named after the operation parameters; the data member for the return value is named `returnValue.`

For example, if we add a new out parameter to `getName`:

Slice

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

The Slice to C++ compiler will generate:

C++

```
class Employees : public virtual Ice::Object
{
public:

    struct GetNameResult
    {
        std::string returnValue;
        std::string email;
    };
    ...
};

class EmployeesPrx : public virtual Ice::ObjectPrx
{
public:

    template<template<typename> class P = std::promise>
    auto getNameAsync(int number, const Ice::Context& ctx = Ice::noExplicitContext)
            -> decltype(std::declval<P<Employees:GetNameResult>>().get_future());

    ...
};
```

You would typically use `auto` to avoid typing the name of this `Result` struct:

C++11

```
auto e = ... // get an Employees proxy
auto fut = e->getNameAsync(99); // get future<Employees::GetNameResult>
```

Callback-Based Async Function

With the callback-based async function, you must provide all the mandatory in-parameters of the operation, followed by a response callback. You can then optionally provide an exception callback and a sent callback.

These callbacks are described below:

- response callback
  The Ice run time calls the response callback to deliver asynchronously the response from a two-way invocation that completes successfully. The signature for this response callback is `std::function<void(`*`return-type`*`, `*`first-out-type`*`, `*`second-out-type`*`...)>`. The response callback for an operation with no return or out parameter has no parameters. Otherwise, all the parameters to this callback function are passed by value, to allow your callback to adopt (move) the memory allocated by the Ice run time (the caller).

- exception callback
  The Ice run time calls the exception callback (when provided) to deliver asynchronously the result of an invocation that completes with an error. This exception callback accepts a single `std::exception_ptr` parameter, passed by value, that can hold any type of exception.

- sent callback
  When you call an `Async` function 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. The Ice run time calls the sent callback (if provided) to notify you that the request has been accepted by the transport. `sent` accepts a single `bool` parameter, set to `true` when the request is sent synchronously, and `false` otherwise.

For example:

C++

```
auto e = ... // get an Employees proxy
e->getNameAsync(99,
                [](string name) { cout << "Employee name is: " << name << endl; },
                [](exception_ptr eptr)
                {
                    try
                    {
                        rethrow_exception(eptr);
                    }
                    catch(const std::exception& ex)
                    {
                        cerr << "Request failed: " << ex.what() << endl;
                    }
                });
```

The Ice run time calls these callbacks using a thread from the communicator's client thread pool, with one exception: the sent callback is called by the thread making the invocation when the request is sent synchronously.

Asynchronous Exception Semantics

If an invocation raises an exception, the exception is reported by the exception callback or by the future, even if the actual error condition for the exception was encountered during the call to the `Async` function ("on the way out"). The advantage of this behavior is that all exception handling is located in the same place (instead of being present twice, once where you call the `Async` function, and again where you retrieve the result) .

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `Async` function throws `CommunicatorDestroyedException`.
  This is necessary because, once the communicator is destroyed, its client thread pool is no longer available.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

Asynchronous Oneway Invocations

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 an `Async` function on a oneway proxy for an operation that returns values or raises a user exception, the `Async` function throws `TwowayOnlyException`.

An `async` oneway invocation does not call the response callback with the callback API; you use the sent callback to make sure the invocation was successfully sent. With the future-based API, the returned future is a `future<void>` and this future is made ready when the invocation is sent.

Canceling an Asynchronous Invocation

The `Async` function with callback parameters returns a cancel function-object (a `std::function<void()>`). You can use this function-object to cancel the invocation, for example:

C++

```
auto e = ... // get an Employees proxy
auto cancel = e->getNameAsync(99, [](string name) { cout << "Employee name is: " << name << endl; });
cancel(); // no longer interested in this name
```

Calling this cancel function-object prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. This cancelation is purely local and has no effect on the server.

Canceling an invocation that has already completed has no effect. Otherwise, a canceled invocation is considered to be completed, meaning the exception callback (if provided) receives an `Ice::InvocationCanceledException`.

Polling for Completion

The future-based `Async` function 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(...);
shared_ptr<FileTransferPrx> ft = ...;
const int chunkSize = ...;

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(...);
shared_ptr<FileTransferPrx> ft = ...;
const int chunkSize = ...;
int offset = 0;

deque<future<void>> results;
const int numRequests = 5;

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

    // Send up to numRequests + 1 chunks asynchronously.
    auto fut = ft->sendAsync(offset, bs);
    offset += bs.size();

    results.push_back(std::move(fut));

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.size() > numRequests)
    {
        results.front().get();
        results.pop_front();
    }
}

// Wait for any remaining requests to complete.
while(!results.empty())
{
    results.front().get();
    results.pop_front();
}
```

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.

Flow Control

Asynchronous method invocations never block the thread that calls the `Async` function : 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. 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.

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 callback 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 example:

C++

```
auto e = ...; // get an Employees proxy

e->getNameAsync(99,
                [](string name) { ... handle name ... },
                [](exception_ptr ex) { ... handle exception ... },
                [](bool) { ... increase sent counter ... });
```

Asynchronous Batch Requests

You can invoke operations via batch 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 an asynchronous proxy method on a batch oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

A batch oneway invocation never calls the response or sent callbacks with the callback API. With the future-based API, the returned future for a batch oneway invocation is always ready and indicates the successful queuing of the batch invocation. The future completes exceptionally if an error occurs before the request is queued.

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 `Async` versions of this function so you can flush batch requests asynchronously:

C++

```
// Future-based function
// with the P = std::promise (the default), it's equivalent to:
// std::future<void> ice_flushBatchRequestsAsync();
//
template<template<typename> class P = std::promise>
auto ice_flushBatchRequestsAsync() -> decltype(std::declval<P<void>>().get_future());

// Callback-based function
//
std::function<void()>
ice_flushBatchRequestsAsync(std::function<void(std::exception_ptr)> ex, std::function<void(bool)> sent =
nullptr);
```

The bool value returned by the future-based function indicates whether the flush was performed synchronously (return value is `true`) or asynchronously (return value is `false`).

Similar `flushBatchRequestsAsync` functions are also available on `Communicator` and `Connection:`

C++

```
// Future-based function
// with the P = std::promise (the default), it's equivalent to:
// std::future<void> flushBatchRequestsAsync(Ice::CompressBatch compress);
//
template<template<typename> class P = std::promise>
auto flushBatchRequestsAsync(Ice::CompressBatch compress) -> decltype(std::declval<P<void>>().get_future())

// Callback-based function
```

```
//
std::function<void()>
flushBatchRequestsAsync(Ice::CompressBatch compress,
                        std::function<void(std::exception_ptr)> exception,
                        std::function<void(bool)> sent = nullptr);
```

As described on the Batched Invocations page, `flushBatchRequests` on `Communicator` and `Connection` flushes only requests made with fixed proxies.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch