Asynchronous Method Invocation (AMI) in Java

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 Java

Consider the following simple Slice definition:

Slice

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

Asynchronous Proxy Methods in Java

In addition to the synchronous proxy methods, `slice2java` generates the following asynchronous proxy methods:

Java

```
public interface EmployeesPrx extends com.zeroc.Ice.ObjectPrx
{
    // ...

    java.util.concurrent.CompletableFuture<java.lang.String> getNameAsync(int number);
    java.util.concurrent.CompletableFuture<java.lang.String> getNameAsync(int number, java.util.Map<String, String> context);
}
```

As you can see, the `getName` Slice operation generates a `getNameAsync` method, along with an overload so that you can pass a per-invocation context.

The `getNameAsync` method sends (or queues) an invocation of `getName`. This method does not block the calling thread. It returns an instance of `java.util.concurrent.CompletableFuture` that you can use in a number of ways, including blocking to obtain the result, configuring an action to be executed when the result becomes available, and canceling the invocation.

Here's an example that calls `getNameAsync`:

Java

```
EmployeesPrx e = ...;
java.util.concurrent.CompletableFuture<String> f = e.getNameAsync(99);

// Continue to do other things here...

String name = f.join();
```

Because `getNameAsync` does not block, the calling thread can do other things while the operation is in progress.

An asynchronous proxy method uses the same parameter mapping as for synchronous operations; the only difference is that the result (if any) is returned in a `CompletableFuture`. An operation that returns no values maps to an asynchronous proxy method that returns `CompletableFuture<Void>`. For example, consider the following operation:

Slice

```
interface Example
{
    double op(int inp1, string inp2, out bool outp1, out long outp2);
}
```

The generated code looks like this:

Java

```
public interface Example
{
    public static class OpResult
    {
        public OpResult();
        public OpResult(double returnValue, boolean outp1, long outp2);

        public double returnValue;
        public boolean outp1;
        public long outp2;
    }

    ...
}

public interface ExamplePrx extends com.zeroc.Ice.ObjectPrx
{
    java.util.concurrent.CompletableFuture<Example.OpResult> opAsync(int inp1, String inp2);
    java.util.concurrent.CompletableFuture<Example.OpResult> opAsync(int inp1, String inp2, java.util.
Map<String, String> context);

    ...
}
```

Now let's call `whenComplete` to demonstrate one way of asynchronously executing an action when the invocation completes:

Java

```
ExamplePrx e = ...;
e.opAsync(5, "demo").whenComplete((result, ex) ->
    {
        if(ex != null)
        {
            // handle exception...
        }
        else
        {
            System.out.println("returnValue = " + result.returnValue);
            System.out.println("outp1 = " + result.outp1);
            System.out.println("outp2 = " + result.outp2);
        }
    });
```

Asynchronous Exception Semantics in Java

If an invocation raises an exception, the exception can be obtained from the future in several ways:

- Call `get` on the future; `get` raises `CompletionException` with the actual exception available via `getCause()`
- Call `join` on the future; `join` raises `ExecutionException` with the actual exception available via `getCause()`
- Use chaining methods such as `exceptionally`, `handle` or `whenComplete` to execute custom actions

The exception is provided by the future, even if the actual error condition for the exception was encountered during the call to the `opAsync` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that handles the future (instead of being present twice, once where the `opAsync` method is called, and again where the future is handled).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- 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.

`InvocationFuture` Class in Java

The `CompletableFuture` object that is returned by asynchronous proxy methods can be down-cast to `InvocationFuture` when an application requires more control over an invocation:

Java

```
package com.zeroc.Ice;

public class InvocationFuture<T> extends ...
{
    public Communicator getCommunicator();
    public Connection getConnection();
    public ObjectPrx getProxy();
    public String getOperation();

    public void waitForCompleted();

    public boolean isSent();
    public void waitForSent();
    public boolean sentSynchronously();

    public CompletableFuture<Boolean> whenSent(java.util.function.BiConsumer<Boolean, ? super Throwable>
action);
    public abstract CompletableFuture<Boolean> whenSentAsync(java.util.function.BiConsumer<Boolean, ? super
Throwable> action);
    public abstract CompletableFuture<Boolean> whenSentAsync(java.util.function.BiConsumer<Boolean, ? super
Throwable> action, Executor executor);
}
```

The methods have the following semantics:

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

- `Connection 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 future is obtained by calling `flushBatchRequestsAsync` on a `Connection` object.

- `ObjectPrx getProxy()`
  This method returns the proxy that was used to call the asynchronous proxy method, or nil if the future was not obtained via an asynchronous proxy invocation.

- `String getOperation()`
  This method returns the name of the operation.

- `void waitForCompleted()`
  This method blocks the caller until the result of an invocation becomes available. Upon return, the standard method `CompletableFuture.isDone()` returns true.

- `boolean isSent()`
  When you call an asynchronous proxy 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 obtain the exception using the standard `CompletableFuture` methods.

- `boolean 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, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

- `CompletableFuture<Boolean> whenSent(BiConsumer<Boolean, ? super Throwable> action)`
  Configures an action to be executed when the request has been successfully written to the client-side transport. The arguments to the action are a boolean indicating whether the request was sent synchronously (see `sentSynchronously` above) and a `Throwable`. The exception argument will be null if the request was sent successfully. The returned stage is completed when the action returns. If the supplied action itself encounters an exception, then the returned stage exceptionally completes with this exception unless this stage also completed exceptionally. If the invocation is already sent at the time `whenSent` is called, the callback method is invoked recursively from the calling thread. Otherwise, the callback method is invoked by an Ice thread (or by a dispatcher if one is configured).
- `CompletableFuture<Boolean> whenSentAsync(BiConsumer<Boolean, ? super Throwable> action)`
  Behaves like `whenSent` except the given action is executed asynchronously using this stage's default asynchronous execution facility.
- `CompletableFuture<Boolean> whenSentAsync(BiConsumer<Boolean, ? super Throwable> action, Executor executor)`
  Behaves like `whenSent` except the given action is executed using the supplied executor.

Due to limitations in Java's generic type system, a regular down-cast from `CompletableFuture<T>` to `InvocationFuture<T>` would cause a compiler error, therefore Ice provides a helper method to perform the conversion for you:

Java

```
package com.zeroc.Ice;

public class Util
{
    static public InvocationFuture getInvocationFuture(java.util.concurrent.CompletableFuture f);

    ...
}
```

See below for sample code that uses this method.

Polling for Completion in Java

The `InvocationFuture` 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:

Java

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;
while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.length;
}
```

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:

Java

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;

LinkedList<InvocationFuture<Void>> results = new LinkedList<InvocationFuture<Void>>();
int numRequests = 5;

while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    CompletableFuture<Void> f = ft.sendAsync(offset, bs);
    offset += bs.length;

    // Wait until this request has been passed to the transport.
    InvocationFuture<Void> i = Util.getInvocationFuture(f);
    i.waitForSent();
    results.add(i);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.size() > numRequests)
    {
        i = results.getFirst();
        results.removeFirst();
        i.join();
    }
}

// Wait for any remaining requests to complete.
while(results.size() > 0)
{
    InvocationFuture<Void> i = results.getFirst();
    results.removeFirst();
    i.join();
}
```

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.

Asynchronous Oneway Invocations in Java

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

The future returned for a oneway invocation completes as soon as the request is successfully written to the client-side transport. The future completes exceptionally if an error occurs before the request is successfully written.

Flow Control in Java

Asynchronous method invocations never block the thread that calls the asynchronous proxy 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, `InvocationFuture.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, `InvocationFuture.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 `InvocationFuture` class 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:

Java

```
ExamplePrx proxy = ...;

CompletableFuture<Result> f = proxy.doSomethingAsync();
InvocationFuture<Result> i = Util.getInvocationFuture(f);
i.whenSent((sentSynchronously, ex) ->
    {
        if(ex != null)
        {
            // handle errors...
        }
        else
        {
            // this request was sent, send another!
        }
    });
```

The `whenSent` method has the following semantics:

- If the Ice run time was able to pass the entire request to the local transport immediately, the action will be invoked from the current thread and the `sentSynchronously` argument will be true.
- If Ice wasn't able to write the entire request without blocking, the action will eventually be invoked from an Ice thread pool thread and the `sentSynchronously` argument will be false.

If you need more control over the execution environment of your action, you can use one of the `whenSentAsync` methods instead. The `sentSynchronously` argument still behaves as described above, but your executor's implementation will determine the threading behavior.

Asynchronous Batch Requests in Java

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`.

The future returned for a batch oneway invocation is always completed 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 asynchronous versions of this method so you can flush batch requests asynchronously.

The proxy method `ice_flushBatchRequestsAsync` flushes any batch requests queued by that proxy. In addition, similar methods are available on the communicator and the `Connection` object that is returned by `InvocationFuture.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Canceling Asynchronous Requests in Java

`CompletableFuture` provides a `cancel` method that you can call to cancel an invocation. If the future hasn't already completed either successfully or exceptionally, cancelling the future causes it to complete with an instance of `java.util.concurrent.CancellationException`.

> Cancellation prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. Cancellation is a local operation and has no effect on the server.

Concurrency Semantics for AMI in Java

For the `CompletableFuture` returned by an asynchronous proxy method, the Ice run time invokes `complete` or `completeExceptionally` from an Ice thread pool thread. The thread in which your action executes depends on the completion status of the future and the manner in which you registered the action. Here are some examples:

- Suppose you configure an action using `whenComplete`. If the future is already complete at the time you call `whenComplete`, the action will execute immediately in the calling thread. If the future is not yet complete when you call `whenComplete`, the action will eventually execute in an Ice thread pool thread.
- Now suppose you configure an action using one of the `whenCompleteAsync` methods. Regardless of the thread in which Ice completes the future, your executor's implementation will determine the thread context in which the action is invoked. The Ice thread pool can be used as an executor; you can obtain the executor by calling the `ice_executor` proxy method and passing it to `whenCompleteAsync`. With the Ice thread pool executor, the action is always queued to be executed by the Ice thread pool. If a dispatcher is configured, the action will be passed to the configured dispatcher by the Ice thread pool thread that dequeues it, otherwise the action will be executed by the Ice thread pool thread that dequeues it.

Refer to the flow control discussion for information about the concurrency semantics of the flow control methods.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch