Asynchronous Method Invocation (AMI) in MATLAB

Previous Next

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 application. 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:

Asynchronous API in MATLAB

Consider the following simple Slice definition:

Slice

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

Asynchronous Proxy Methods in MATLAB

In addition to the synchronous proxy method, `slice2matlab` generates the following asynchronous proxy method:

MATLAB

```
classdef EmployeesPrx < Ice.ObjectPrx
    methods
        function result = getName(obj, number, varargin)  % Synchronous method
            ...
        end
        function future = getNameAsync(obj, number, varargin) % Asynchronous method
            ...
        end
    end
    ...
end
```

As you can see, the `getName` Slice operation generates a `getNameAsync` method that optionally accepts a per-invocation context.

The `getNameAsync` method sends (or queues) an invocation of `getName`. This method does not block the application. It returns an instance of `Ice.Future` that you can use in a number of ways, including blocking to obtain the result, querying its state, and canceling the invocation.

Here's an example that calls `getNameAsync`:

MATLAB

```
e = ...; % Get EmployeesPrx proxy
future = e.getNameAsync(99);

% Continue to do other things here...

name = f.fetchOutputs();
```

Because `getNameAsync` does not block, the application 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 obtained from the future. 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:

MATLAB

```
classdef ExamplePrx < Ice.ObjectPrx
    methods
        function future = opAsync(obj, inp1, inp2, varargin)
            ...
        end
        ...
    end
    ...
end
```

Now let's call `fetchOutputs` to demonstrate how to retrieve the results when the invocation completes:

MATLAB

```
e = ...; % Get EmployeesPrx proxy
future = e.opAsync(5, 'demo');
...
[r, outp1, outp2] = future.fetchOutputs();
```

Back to Top ^

Asynchronous Exception Semantics in MATLAB

If an invocation raises an exception, the exception will be thrown when the application calls `fetchOutputs` on the future. 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 `CommunicatorDestroyedExceptio` 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.

Back to Top ^

`Future` Class in MATLAB

The `Future` object that is returned by asynchronous proxy methods has an API that resembles MATLAB's `parallel.Future` class:

MATLAB

```
classdef Future < ...
    methods
        function ok = wait(obj, state, timeout)
        function varargout = fetchOutputs(obj)
        function cancel(obj)
    end
    properties(SetAccess=private) % Read only properties
        ID
        NumOutputArguments
        Operation
        Read
        State
    end
end
```

The members have the following semantics:

- `wait()`
  This method blocks until the invocation completes and returns `true` if it completed successfully or `false` if it failed.
- `wait(state)`
  This method blocks until the desired state is reached (see the description of the `State` property below). For example, calling `future.wait('finished')` is equivalent to calling `future.wait()`. The method returns `true` if the desired state was reached and no exception has occurred, or `false` otherwise.
- `wait(state, timeout)`
  This method blocks for a maximum of `timeout` seconds until the desired state is reached, where `timeout` is a double value. The method returns `true` if the desired state was reached and no exception has occurred, or `false` otherwise.
- `fetchOutputs()`
  This method blocks until the invocation completes. If it completed successfully, `fetchOutputs` returns the results (if any). If the invocation failed, `fetchOutputs` raises the exception. This method can only be invoked once.
- `cancel()`
  If the invocation hasn't already completed either successfully or exceptionally, cancelling the future causes it to complete with an instance of `InvocationCanceledException`. 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.
- `ID`
  A numeric value that uniquely identifies the invocation.
- `NumOutputArguments`
  A numeric value denoting how many results the invocation will return upon successful completion.
- `Operation`
  The name of the operation that was invoked.
- `Read`
  A logical value indicating whether the results have already been obtained via `fetchOutputs`.
- `State`
  A string value indicating the current state of the invocation. For a twoway invocation, the property value transitions from `running` to `sent` to `finished`. For a oneway, datagram, or batch invocation, the property value transitions from `running` to `finished`.

Back to Top ^

Asynchronous Oneway Invocations in MATLAB

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.

Back to Top ^

Asynchronous Batch Requests in MATLAB

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 application 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. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Back to Top ^

See Also

- Request Contexts
- Batched Invocations

Previous Next