Overview of Deprecated AMI Mapping

On this page:

Motivation for AMI

Modern middleware technologies attempt to ease the programmer's transition to distributed application development by making remote invocations as easy to use as traditional method calls: a method is invoked on an object and, when the method completes, the results are returned or an exception is raised. Of course, in a distributed system the object's implementation may reside on another host, and consequently there are some semantic differences that the programmer must be aware of, such as the overhead of remote invocations and the possibility of network-related errors. Despite those issues, the programmer's experience with object-oriented programming is still relevant, and this *synchronous* programming model, in which the calling thread is blocked until the operation returns, is familiar and easily understood.

Ice is inherently an asynchronous middleware platform that simulates synchronous behavior for the benefit of applications (and their programmers). When an Ice application makes a synchronous twoway invocation on a proxy for a remote object, the operation's in parameters are marshaled into a message that is written to a transport, and the calling thread is blocked in order to simulate a synchronous method call. Meanwhile, the Ice run time operates in the background, processing messages until the desired reply is received and the calling thread can be unblocked to unmarshal the results.

There are many cases, however, in which the blocking nature of synchronous programming is too restrictive. For example, the application may have useful work it can do while it awaits the response to a remote invocation; using a synchronous invocation in this case forces the application to either postpone the work until the response is received, or perform this work in a separate thread. When neither of these alternatives are acceptable, the asynchronous facilities provided with Ice are an effective solution for improving performance and scalability, or simplifying complex application tasks.

Simulating AMI using Oneway Invocations

Before we get into the details of the AMI facilities that Ice provides, let's explore the idea of using oneway invocations to achieve similar results. As an example, consider an application with a graphical user interface. This application typically must avoid blocking the window system's event dispatch thread because blocking makes the application unresponsive to user commands. In this situation, making a synchronous remote invocation is asking for trouble.

The application could attempt to avoid this situation using oneway invocations, which by definition cannot return a value or have any `out` parameters. Since the Ice run time does not expect a reply, the invocation blocks only as long as it takes to establish a connection (if necessary), marshal the request, and copy the message into the local transport buffer. However, these network activities may still block. Furthermore, the use of oneway invocations may require unacceptable changes to the interface definitions. For example, a twoway invocation that returns results or raises user exceptions must be converted into at least two operations: one for the client to invoke with oneway semantics that contains only in parameters, and one (or more) for the server to invoke to notify the client of the results.

To illustrate these changes, suppose that we have the following Slice definition:

Slice

```
interface I {
  int op(string s, out long l);
};
```

In its current form, the operation `op` is not suitable for a oneway invocation because it has an `out` parameter and a non-`void` return type. In order to accommodate a oneway invocation of `op`, we can change the Slice definitions as shown below:

Slice

```
interface ICallback {
  void opResults(int result, long l);
};

interface I {
  void op(ICallback* cb, string s);
};
```

We made several modifications to the original definition:

- We added the interface `ICallback`, containing an operation `opResults` whose arguments represent the results of the original twoway operation. The server invokes this operation to notify the client of the completion of the operation.
- We modified `I::op` to be compliant with oneway semantics: it now has a `void` return type, and takes only in parameters.
- We added a parameter to `I::op` that allows the client to supply a proxy for its callback object.

As you can see, we have made significant changes to our interface definitions to accommodate the implementation requirements of the client. One ramification of these changes is that the client must now also be a server, because it must create an instance of `ICallback` and register it with an object adapter in order to receive notifications of completed operations.

A more severe ramification, however, is the impact these changes have on the type system, and therefore on the server. Whether a client invokes an operation synchronously or asynchronously should be irrelevant to the server; this is an artifact of behavior that should have no impact on the type system. By changing the type system as shown above, we have tightly coupled the server to the client, and eliminated the ability for `op` to be invoked synchronously.

To make matters even worse, consider what would happen if `op` could raise user exceptions. In this case, `ICallback` would have to be expanded with additional operations that allow the server to notify the client of the occurrence of each exception. Since exceptions cannot be used as parameter or member types in Slice, this quickly becomes a difficult endeavor, and the results are likely to be equally difficult to use.

At this point, you will hopefully agree that this technique is flawed in many ways, so why do we bother describing it in such detail? The reason is that the Ice implementation of AMI uses a strategy similar to the one described above, with several important differences:

1. No changes to the type system are required in order to use AMI. The on-the-wire representation of the data is identical, therefore synchronous and asynchronous clients and servers can coexist in the same system, using the same operations.
2. The AMI solution accommodates exceptions in a reasonable way.
3. Using AMI does not require the client to also be a server.
4. Ice guarantees that AMI requests never block the calling thread.

AMI Semantics

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, in the case of a twoway invocation, is notified when the reply eventually arrives. Notification occurs via a callback to an application-supplied programming-language object.

Polling for a response is not supported in the deprecated AMI mapping, but polling is supported in the new API.

The use of an asynchronous model does not affect what is sent "on the wire." Specifically, the invocation model used by a client is transparent to the server, and the dispatch model used by a server is transparent to the client. Therefore, a server has no way to distinguish a client's synchronous invocation from an asynchronous invocation, and a client has no way to distinguish a server's synchronous reply from an asynchronous reply.

Controlling AMI Code Generation using Metadata

A programmer indicates a desire to use an asynchronous model (AMI, AMD, or both) by annotating Slice definitions with metadata. The programmer can specify this metadata at two levels: for an interface or class, or for an individual operation. If specified for an interface or class, then asynchronous support is generated for all of its operations. Alternatively, if asynchronous support is needed only for certain operations, then the generated code can be minimized by specifying the metadata only for those operations that require it.

Synchronous invocation methods are always generated in a proxy; specifying AMI metadata merely adds asynchronous invocation methods. In contrast, specifying AMD metadata causes the synchronous dispatch methods to be *replaced* with their asynchronous counterparts. This semantic difference between AMI and AMD is ultimately practical: it is beneficial to provide a client with synchronous and asynchronous versions of an invocation method, but doing the equivalent in a server would require the programmer to implement both versions of the dispatch method, which has no tangible benefits and several potential pitfalls.

Consider the following Slice definitions:

Slice

```
["ami"] interface I {
    bool isValid();
    float computeRate();
};

interface J {
  ["amd"]        void startProcess();
  ["ami", "amd"] int endProcess();
};
```

In this example, all proxy methods of interface `I` are generated with support for synchronous and asynchronous invocations. In interface `J`, the `startProcess` operation uses asynchronous dispatch, and the `endProcess` operation supports asynchronous invocation and dispatch.

Specifying metadata at the operation level, rather than at the interface or class level, not only minimizes the amount of generated code, but more importantly, it minimizes complexity. Although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations for which it provides a particular advantage, while using the simpler synchronous model for the rest.

AMI Interfaces

AMI operations have the same semantics in all of the language mappings that support asynchronous invocations. This section provides a language-independent introduction to the AMI model.

Proxy Method for AMI Interfaces

Annotating a Slice operation with the AMI metadata tag does not prevent an application from invoking that operation using the traditional synchronous model. Rather, the presence of the metadata extends the proxy with an asynchronous version of the operation, so that invocations can be made using either model.

The asynchronous operation never blocks the calling thread. If the message cannot be accepted into the local transport buffer without blocking, the Ice run time queues the request and immediately returns control to the calling thread.

The parameters of the asynchronous operation are modified similar to our earlier example: the first argument is a callback object (described below), followed by any `in` parameters in the order of declaration. The operation's return value and `out` parameters, if any, are passed to the callback object when the response is received.

The asynchronous operation only raises `CommunicatorDestroyedException` directly; all other exceptions are reported to the callback object.

Finally, the return value of the asynchronous operation is a boolean that indicates whether the Ice run time was able to send the request synchronously; that is, whether the entire message was immediately accepted by the local transport buffer. An application can use this value to implement flow control.

Callback Object for AMI Interfaces

The asynchronous operation requires the application to supply a callback object as the first argument. This object is an instance of an application-defined class; in strongly-typed languages this class must inherit from a superclass generated by the Slice compiler. In contrast to the oneway example, the callback object is a purely local object that is invoked by the Ice run time in the client, and not by the remote server.

The Ice run time always invokes methods of the callback object from a thread in an Ice thread pool, and never from the thread that is invoking the asynchronous operation. Exceptions raised by a callback object are ignored but may cause the Ice run time to log a warning message depending on the value of the `Ice.Warn.AMICallback` property.

The callback class must define the methods `ice_response` and `ice_exception`:

- `ice_response`
  Invoked by the Ice run time to supply the results of a successful twoway invocation; this method is not invoked for oneway invocations. The

arguments to `ice_response` consist of the return value (if the operation returns a non-`void` type) followed by any `out` parameters in the order of declaration.

- `ice_exception`
  Handles errors that occur during the invocation. The only exception that can be raised to the thread invoking the asynchronous operation is `CommunicatorDestroyedException`; all other errors, including user exceptions, are passed to the callback object via its `ice_exception` method. In the case of a oneway invocation, `ice_exception` is only invoked if an error occurs before the request is sent.

For an asynchronous invocation, the Ice run time calls `ice_response` or `ice_exception`, but not both. It is possible for one of these methods to be called before control returns to the thread that is invoking the operation.

A callback object may optionally define a third method, `ice_sent`:

- `ice_sent`
  Invoked when the entire message has been passed to the local transport buffer to provide flow control functionality. The Ice run time does not invoke `ice_sent` if the asynchronous operation returned true to indicate that the message was sent synchronously. An application must make no assumptions about the order of invocations on a callback object; `ice_sent` can be called before, after, or concurrently with `ice_response` or `ice_exception`.

See Also

- Metadata
- Oneway Invocations
- Advanced Topics for Deprecated AMI Mapping