Object Interconnections

Programming Asynchronous Method Invocations with CORBA Messaging
(Column 16)

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Steve Vinoski
vinoski@iona.com
IONA Technologies, Inc.
60 Aberdeen Ave., Cambridge, MA 02138

This column will appear in the February 1999 issue of the SIGS C++ Report magazine.

1 Introduction

Welcome to our continuing coverage of asynchronous messaging and the new CORBA Messaging specification [1]. Our previous column presented an overview of the specification. It also outlined how the Messaging specification alleviates the tedium of programming with deferred synchronous operations via the Dynamic Invocation Interface (DII) and avoids the weak reliability semantics of oneway operations.

In this column, we focus on asynchronous method invocation (AMI), which is a core part of the new CORBA Messaging specification. A key feature of CORBA AMI is that operations can be invoked asynchronously using the *static invocation interface* (SII), thereby eliminating much of the complexity inherent in the DII deferred synchronous model. This column illustrates how to write CORBA applications using the two AMI programming models:

Polling model: In this model, each asynchronous two-way invocation returns a `Poller` *valuetype*, which is a new IDL type introduced by the new *Objects-by-Value* (OBV) specification [2]. A `valuetype` is very much like a C++ or Java class in that it has both data members and methods, which when invoked are just local C++ member function calls and not distributed CORBA operation invocations.

The client can use the `Poller` methods to check the status of the request and to obtain the value of the reply from the server. If the server hasn't returned the reply yet, the client can elect to block awaiting its arrival, just as with the DII deferred synchronous mode shown in our previous column. Alternatively, the client can return to the calling thread immediately and check on the `Poller` later when convenient.

Callback model: In this model, the client passes an object reference for a `ReplyHandler` object as a parameter when it invokes a two-way asynchronous operation on a server. When the server responds, the client ORB receives the response and dispatches it to the appropriate C++ method on the `ReplyHandler` servant so the client can handle the reply. In other words, the ORB turns the response into a request on the client's `ReplyHandler`.

In general, the callback model is more efficient than the polling model because the client need not poll for results. However, it forces clients to behave as servers, which increases the complexity of certain applications, particularly "pure" clients.

2 Using CORBA Asynchronous Method Invocations (AMI)

To illustrate CORBA AMI, we use the following `Quoter` interface, which we've used as a running example in many of our previous columns.

```
module Stock
{
  // Requested stock does not exist.
  exception Invalid_Stock {};

  interface Quoter {
    // Two-way operation to retrieve current
    // stock value.
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
  };

  // ...
}
```

To make our example more interesting, we'll write a client `get_stock_quotes` helper function using three implementations: (1) a synchronous model, (2) an asynchronous polling model, and (3) an asynchronous callback model. The `get_stock_quotes` function will invoke the two-way `get_quote` operation defined in the `Quoter` interface to retrieve the current stock value for various publicly traded ORB vendors. We'll use the following global variables for each implementation:

```
// NASDAQ abbreviations for ORB vendors.
static const char *orbs[] =
{
  "IONAY" // IONA Orbix
  "BEAS"  // BEA systems M3
  "INPR"  // Inprise VisiBroker
  "IBM"   // IBM Component Broker
  "SUNW"  // Sun/Chorus COOL
}

// Set the max number of ORBs.
static const int MAX_ORBS =
  sizeof (orbs) / sizeof (*orbs);

// Keep track of the asynchronous
// reply count.
static int reply_count = MAX_ORBS;
```

We show each implementation below.

2.1 Synchronous Model

Implementation: Assuming we've got an object reference of type `Quoter`, here's a quick refresher on the steps required to write a client `get_stock_quotes` method that uses the synchronous model:

1. Generate stubs from the IDL file: In this step, we run the `Stock module` through a conventional IDL compiler. The IDL compiler automatically generates the stubs that implement the SII C++ mapping.

2. Write the client using stubs: In this step, we simply use the stubs generated by the compiler to write our client `get_stock_quotes` method, as follows:

```
void get_stock_quotes
  (CORBA::ORB_ptr orb,
   Stock::Quoter_ptr quoter_ref)
{
  // Make synchronous two-way calls.
  for (int i = 0; i < MAX_ORBS; i++) {
    CORBA::Long value =
      quoter_ref->get_quote (orbs[i]);
    cout << "Current value of "
         << orbs[i] << " stock: "
         << value << endl;
  }

  // ...
}
```

Each call to the `get_quote` operation causes the stub to transmit the request to the object. The client then blocks synchronously in the stub waiting for the reply to return from the server.

Evaluating the synchronous model: This code is obvious and natural to most C++ programmers because it uses two-way synchronous calls. However, the overall time required to complete the `for` loop will depend largely on the latency of the longest two-way call. Moreover, if earlier calls take a long time to return, subsequent calls will be delayed, even if they could return immediately.

While the calling thread is blocked waiting for each response, there is nothing else that it can do. Moreover, if the client is single-threaded, this means that the whole process is blocked while waiting for each reply. Many types of applications, particularly real-time applications that monitor embedded control systems [3], cannot afford to block an entire process while waiting for a reply.

To work around the limitations with the synchronous model shown above, we reimplement the client `get_stock_quotes` function below using the asynchronous polling model defined in the CORBA Messaging AMI specification.

2.2 Asynchronous Polling Model

Implementation: The AMI polling model implementation for the client `get_stock_quotes` function requires the following steps:

1. Generate "implied IDL" for polling signatures: In this step, we run the `Stock module` through an IDL compiler that supports the C++ mapping for the AMI polling model. Such a compiler maps operations and attributes in each IDL `interface` to *implied IDL*. The term *implied IDL* refers to the fact that the IDL compiler "logically" generates additional IDL based on the standard IDL declarations fed into it, and then compiles the original IDL and the implied IDL into stubs and skeletons.

In the implied IDL mapping for the polling model, the IDL compiler generates polling operation and attribute names that are prefixed by "`sendp_`." In general, `in` and `inout` parameters in each original IDL operation map to `in` parameters in each `sendp_` operation.

In the stock quote example, the automatically-generated implied polling operation is called `sendp_get_quote` and has the following signature:

```
namespace Stock
{
  class Quoter
    : public virtual CORBA::Object
  {
  public:
    Stock::AMI_QuoterPoller *
     sendp_get_quote (const char *stockname);

  // ...
  };
};
```

Rather than blocking until the reply arrives, the `sendp_get_quote` operation returns a pointer to a `Stock::AMI_QuoterPoller`, which is defined by the following automatically-generated implied class:

```
namespace Stock
{
  class AMI_QuoterPoller
    : public Messaging::Poller
  {
  public:
    virtual void get_quote
     (CORBA::ULong timeout,
      CORBA::Long_out ami_return_val);
  };
}
```

The client can subsequently query an `AMI_QuoterPoller` object to retrieve the stock value when the reply arrives from the server. Because `AMI_QuoterPoller` is a `valuetype`, an invocation of the `get_quote` method on an `AMI_QuoterPoller` object is always *collocated*, *i.e.*, the call is invoked locally and does not go across the network.

In general, `inout` and `out` parameters and return values for each operation in an IDL interface `Foo` map to `out` parameters in each method in the automatically generated `AMI_FooPoller` class. Moreover, each method can be given a timeout to bound the amount of time the client is willing to wait for the reply to return. If the operation invocation results in an exception, its corresponding `AMI_FooPoller` method throws that exception when you invoke it.

2. Rewrite the client to use `Pollers`: In this step, we reimplement the client `get_stock_quotes` function to use the generated `sendp_get_quote` operation, which invokes the calls asynchronously, as follows:

```
void get_stock_quotes
  (CORBA::ORB_ptr orb,
   Stock::Quoter_ptr quoter_ref)
{
  Stock::AMI_QuoterPoller *pollers[MAX_ORBS];
  int i;

  // Make asynchronous two-way calls using
  // the polling model.
  for (i = 0; i < MAX_ORBS; i++)
    pollers[i] =
      quoter_ref->sendp_get_quote (orbs[i]);
```

Once all the calls have been invoked asynchronously, the client can query the `Poller` objects for the replies. In this example, the `get_stock_quotes` function will first try to obtain all the replies without blocking, *i.e.*, it will perform a "nonblocking poll" by setting the timeout parameter to 0, as follows:

```
  // Set to the minimum timeout value,
  // i.e., "return immediately".
  CORBA::ULong min_timeout = 0;

  // Obtain the results via
  // "immediate polling".
  for (i = 0; i < MAX_ORBS; i++) {
    try {
      CORBA::Long value;
      // Don't block if the result isn't
      // ready.
      pollers[i]->get_quote (min_timeout,
                             value);
      cout << "Current value of "
           << orbs[i] << " stock: "
           << value << endl;
      // Zero-out the ORB name
      // once we get a return value.
      orbs[i] = 0;
      reply_count--;
    }
    // Catch exception indicating
    // response is not yet available.
    catch (const CORBA::NO_RESPONSE &)
    {}
  }
```

Only if all the replies haven't arrived, *i.e.*, the `reply_count` is > 0, do we actually block the client by setting the timeout parameter to -1, as follows:

```
  // Set to the larger timeout value,
  // i.e., "block forever".
  CORBA::ULong max_timeout =
    CORBA::ULong (-1);

  // Obtain any remaining results via
  // "indefinite polling".
  for (i = 0; i < MAX_ORBS; i++) {
    // Skip replies we've already obtained.
    if (orbs[i] == 0)
      continue;
      CORBA::Long value;
      // Block indefinitely until the result
      // is ready.
      pollers[i]->get_quote (max_timeout,
                             value);
      cout << "Current value of "
           << orbs[i] << " stock: "
           << value << endl;
      // Zero-out the ORB name
      // once we get a return value.
      orbs[i] = 0;
    }

  // ...
}
```

In this implementation, the `get_stock_quotes` function does nothing other than poll for the results. Programs that use the AMI polling model would normally perform other processing, however, *e.g.*, they would service a GUI or update a sensor reading in between calls to the `Pollers`.

Evaluating the AMI polling model: The AMI polling model requires programmers to write more C++ code than they do for synchronous two-way calls. For instance, the AMI polling implementation of `get_stock_quotes` must keep track of the `AMI_QuoterPoller` results returned from the `sendp_get_quote` call. Once all the calls are invoked, programmers must then query the `Pollers` explicitly to retrieve the reply. In contrast, conventional synchronous two-way CORBA calls wait for replies in their generated stubs, which alleviates the need for application polling.

An implementation-related drawback to using AMI polling is that it requires you to use Objects-by-Value (OBV) in your application. The OBV specification is overly complicated and has no track record of success in any real-world applications. Unlike most OMG specifications, OBV was mostly invented on-the-fly, with the submitters often making significant changes to the specification only hours before its revision deadlines passed. Because of the low quality and high complexity of the OBV specification, it will take a long time before ORBs properly support it. You may want to avoid it until ORB vendors and the OMG can work all the kinks out of it.

Another drawback with the AMI polling model is that it doesn't really solve the inefficiency of waiting for long latency calls to complete, which was a problem with the synchronous `get_stock_quotes` implementation shown in Section 2.1. In both cases, one long-running call could unduly delay the client from completing other calls. To work around this problem, we'll next implement the client `get_stock_quotes` function using the AMI *callback* model instead of the polling model.[1]

2.3 Asynchronous Callback Model

Implementation: The AMI callback model implementation for the client `get_stock_quotes` function requires

[1] Another way to work around this problem is to use `PollableSets`, which we'll discuss in a subsequent column.

the following steps:

1. Generate implied IDL for callback signatures: In this step, we run the `Stock` `module` through an IDL compiler that supports the C++ mapping for the AMI callback model. Such a compiler will map operations and attributes in each IDL `interface` to implied IDL, as described in Section 2.2. In the implied IDL mapping for the callback model, operation and attribute names are prefixed by "`sendc_`."

As with the polling model, `in` and `inout` parameters in each operation in an IDL interface `Foo` map to `in` parameters in each `sendc_` operation. In addition, the first parameter of each `sendc_` method is a callback of type `AMI_FooHandler_ptr` that is implicitly registered with the client ORB after the call is made. This callback will be invoked by the ORB after the reply returns from the server.

In the stock quote example, the implied callback operation is called `sendc_get_quote` and has the following signature:

```
namespace Stock
{
  class Quoter
    : public virtual CORBA::Object
  {
  public:
    void sendc_get_quote
      (Stock::AMI_QuoterHandler_ptr
       const char *stockname);

  // ...
  };
};
```

The client application is responsible for providing an object that implements the callback. It passes the object reference for this object as the first parameter to the `sendc_get_quote` function. Rather than blocking until the reply arrives, the `sendc_get_quote` operation returns immediately.

2. Implement the ReplyHandler servant: In this step, the client developer must implement a `ReplyHandler` servant. This servant inherits from the `POA_Stock::AMI_QuoterHandler` implied IDL skeleton, which is generated automatically by the IDL compiler, as follows:

```
namespace POA_Stock
{
  class AMI_QuoterHandler
    : public POA_Messaging::ReplyHandler
  {
  public:
    virtual void get_quote
     (CORBA::Long l) = 0;
    virtual void get_quote_excep
     (Stock::AMI_QuoterExceptionHolder *excep)
     = 0;
  };
}
```

The `get_quote` operation handles normal replies, whereas the `get_quote_excep` operation handles exceptional replies. For example, if the client invokes `get_quote` and passes a valid stock name, the client ORB will invoke the `AMI_QuoterHandler::get_quote` to handle the reply, passing in the value of the requested stock. Conversely, if the client passes an unknown stock name, the reply must consist of the `Stock::Invalid_Stock` exception.

For synchronous calls, the ORB propagates exceptions up the runtime stack to the client function that invoked the operation. For asynchronous calls, however, the ORB can't reliably propagate the exception up the stack because the reply may return to a different context than the one in which the original request was made. By calling `get_quote_excep`, therefore, the ORB signifies to the client application that its original request raised an exception.
For our `get_stock_quotes` example, the servant implementation is defined as follows:

```
class My_Async_Stock_Handler
  : public POA_Stock::AMI_QuoterHandler
{
public:
  My_Async_Stock_Handler (const char *stockname)
    : stockname_ (CORBA::string_dup (stockname))
  { }

  ~My_Async_Stock_Handler (void) { }

  virtual void get_quote (CORBA::Long value)
    throw (CORBA::SystemException) {
    cout << "Current value of "
         << stockname_ << " stock: "
         << value << endl;

    // Decrement the number of replies.
    reply_count--;
  }

  virtual void get_quote_excep
    (Stock::AMI_QuoterExceptionHolder *excep)
    throw (CORBA::SystemException,
           Stock::Invalid_Stock) {
    try {
      excep->raise_get_quote ();
    } catch (const Stock::Invalid_Stock &)
    {
      cerr << stockname_ << " is not valid!"
           << endl;
    } catch (const CORBA::SystemException &ex)
    {
      cerr << "get_quote() raised " << ex
           << " for " << stockname_ << endl;
    }

    // Decrement the number of replies.
    reply_count--;
  }

private:
  CORBA::String_var stockname_;
};
```

When the reply for an invocation of the `get_quote` operation returns successfully, the ORB and the POA dispatch it to the servant's `get_quote` reply handler method, passing the `CORBA::Long` return value as an `in` argument.

Unlike the reply for a synchronous invocation, the reply for an asynchronous invocation is not received in the calling context. The stock name we passed when we originally called the `get_quote` operation is not passed to the servant's method because it's part of the request, not part of the reply. Therefore, whenever a `ReplyHandler`'s actions are based on its input arguments, such as in our example where we print the name of the stock, you must provide those input values explicitly to the `ReplyHandler` servant. We do this by passing the stock name to the servant's constructor.

3. Rewrite the client to use callbacks: This step involves reimplementing the client `get_stock_quotes` function. It uses the generated `sendc_get_quote` operation to invoke the calls asynchronously and initializes the `ReplyHandler` callbacks, as follows:

```
void get_stock_quotes
  (CORBA::ORB_ptr orb,
   Stock::Quoter_ptr quoter_ref)
{
  My_Async_Stock_Handler *
    handlers[MAX_ORBS];
  Stock::AMI_QuoterHandler_var
    handler_refs[MAX_ORBS];
  int i;

  // Initialize ReplyHandler servants.
  for (i = 0; i < MAX_ORBS; i++)
    handlers[i] =
      new My_Async_Stock_Handler (orbs[i]);

  // Initialize ReplyHandler object refs.
  for (i = 0; i < MAX_ORBS; i++)
    handler_refs[i] = handlers[i]->_this ();

  // Make asynchronous two-way calls using
  // the callback model.
  for (i = 0; i < MAX_ORBS; i++)
    quoter_ref->sendc_get_quote
      (handler_refs[i],
       orbs[i]);
```

Once the asynchronous calls are all invoked, the client can simply wait in the ORB's event loop for replies to arrive, as follows:

```
  // Callbacks are invoked during the ORB's
  // event loop processing.  We'll keep
  // iterating until all replies have been
  // received.
  while (reply_count > 0)
    if (orb->work_pending ())
      orb->perform_work ();

  // ...
}
```

Because this client runs an ORB event loop, it effectively plays the role of both a server *and* a client. As before, the `get_stock_quotes` function doesn't do anything other than wait for result callbacks, although the `perform_work` method can dispatch other CORBA requests, as well. Applications that use the AMI callback model would normally perform other processing in between calls to `perform_work`, however, and thus would avoid busy loops like the one shown here. It is important to note that `perform_work` is essentially a nonblocking call. Unlike the blocking `ORB::run` function we've used in previous columns, `perform_work` carries out an implementation-defined unit of work (if any) and then returns. The `work_pending` function returns true only when the ORB has work to do.

Evaluating the AMI callback model: A benefit of the AMI callback model is that it helps solve the problems with waiting efficiently for long latency calls to complete. With this technique, long-running calls don't interfere with other calls. It also allows single-threaded applications to avoid blocking while waiting for responses. This feature potentially makes programming easier and avoids the need to determine the best threading model to use for the application.

We described problems with callback-based servers in a previous column [4]. For instance, we pointed out problems with distributed callbacks that are caused by the server having to manage and keep track of many callback objects, their registrations, and the data sent to them. Fortunately, these problems do not arise with the AMI callback model because the server ORB treats the asynchronous request as it does a synchronous request, *i.e.*, it dispatches the request and returns the reply. The client ORB, not the server, handles the asynchronous aspects of the reply. In fact, existing CORBA servers need not be changed at all to handle AMI requests.

The AMI callback model requires client programmers to write more code than either the synchronous model or the AMI polling model, however. In particular, client programmers must write the `My_Async_Stock_Handler` servant, as well as the associated client event loop code to manage the asynchronous replies.

For applications that are already event-driven, this extra code may not incur much additional effort. However, for "pure" clients it can be inconvenient to restructure the code to support callbacks. For instance, memory management is more complicated, as is scoping of variables needed in the callback methods. Our simple example solved the latter problem by defining several global variables. However, the use of global variables generally yields overly coupled applications.

Therefore, despite being simpler than the equivalent DII code, the AMI programming model is not as simple as that of synchronous invocations. In fact, asynchronous applications can be difficult to understand and maintain because their code structures do not represent their calling patterns [5]. In addition, handling exceptions by writing separate methods is nonintuitive.

3 Concluding Remarks

This column illustrated how to write C++ programs that use the polling and callback models defined in the asynchronous method invocation (AMI) section of the new CORBA Messaging specification. An important consequence of both the callback and polling models is that multiple two-way AMIs can be processed asynchronously within a single thread of control. This design simplifies the need for concurrency control that would otherwise be required if multiple threads were used to process multiple two-way synchronous calls simultaneously.

Interestingly, the CORBA Messaging specification allows clients to use the AMI models without requiring any modifications to servers. That's because the CORBA Messaging specification treats asynchronous invocations as a clientside language mapping issue. Our next column will explore several other key aspects of the Messaging specification, such as the *interoperable routing protocol* (IRP) and *time-independent invocations* (TII).

As always, if you have any questions about the material we covered in this column or in any previous ones, please email us at `object_connect@cs.wustl.edu`.

References

[1] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[2] Object Management Group, *Objects-by-Value*, OMG Document orbos/98-01-18 ed., January 1998.

[3] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[4] D. Schmidt and S. Vinoski, "Distributed Callbacks and Decoupled Communication in CORBA," *C++ Report*, vol. 8, October 1996.

[5] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.