Object Interconnections

An Introduction to CORBA Messaging
(Column 15)

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

Steve Vinoski
vinoski@iona.com
IONA Technologies, I nc.
60 Aberd een Ave., Cambridge, MA 02138

This column will appear in the November/December 1998 issue of the SIGS C++ Report magazine.

1 Introduction

In this column we begin our coverage of asynchronous messaging and the new CORBA Messaging specification [1]. This specification introduces a number of long-awaited features into CORBA, including asynchronous method invocation (AMI), time-independent invocation (TII), and general messaging quality of service (QoS) policies. These new features greatly enhance the standard set of request/response communication models that CORBA provides.

This column briefly describes the existing CORBA communication models and illustrates their limitations. We then present an overview of the CORBA Messaging specification and outline how it alleviates limitations with the current CORBA specification.

2 An Overview of CORBA Communication Models

Prior to the Messaging specification, CORBA provided three communication models:

Synchronous twoway: In this model, a client sends a twoway request to a target object and waits for the object to return the response. While it is waiting, the client thread that invoked the request is blocked and can't perform any other processing. Thus, a single-threaded client can be completely blocked while waiting for a response, which may be unsatisfactory for certain types of performance-constrainted applications [2].

Oneway: A oneway invocation is composed of only a request, with no response. The creators of the first version of CORBA intended ORBs to deliver oneways over unreliable transports and protocols such as the User Datagram Protocol (UDP). However, most ORBs implement oneways over TCP, as required by the standard IIOP protocol.

Deferred synchronous: In this model, a client sends a request to a target object and then continues its own processing. Unlike how synchronous twoway requests are handled, the client ORB does not explicitly block the calling thread until the response arrives. Instead, the client can later either (1) poll to see if the target object has returned a response or (2) it can perform a separate blocking call to wait for the response.[1] You can only use the deferred synchronous request model if you invoke your requests using the Dynamic Invocation Interface (DII), however.

CORBA specifies that oneway invocations have "best effort" semantics. Thus, the ORB need not report an error if it is unable to deliver the oneway. In contrast, CORBA guarantees (1) *exactly-once* delivery of synchronous twoway requests if the ORB does not experience errors or exceptions during delivery and (2) *at-most-once* delivery if errors occur or if exceptions are raised. The guarantees for deferred synchronous twoway requests are identical to those for synchronous twoway requests.

While the synchronous request model is pretty straightforward, both the deferred synchronous and oneway models suffer from the drawbacks we describe in the following section.

3 Limitations with DII and Oneway Operations

3.1 The Tedium of the DII

As we stated above, the deferred synchronous model is available only through the DII. In a statically-typed, compiled language like C++, however, the DII is tedious to use, as we show below.

DII example: To illustrate the tedium of the DII, let's look at the `Quoter` interface we've used as a running example in many previous columns [3, 4, 5]:

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

  interface Quoter {
    long get_quote (in string stock_name)
```

[1] It is also possible to send oneway requests using the deferred synchronous model.

```
    raises (Invalid_Stock);
  };

  // ...
}
```

Assuming we have an object reference of type `Quoter`, invoking the `get_quote` operation using the static invocation interface (SII) is trivial:

```
Stock::Quoter_var quoter_ref = // get Quoter reference

CORBA::Long value =
  quoter_ref->get_quote ("IONAY");

cout << "Current value of IONA stock: "
     << value << endl;
```

This code is obvious and natural to most C++ programmers.

Now, instead of using SII, let's invoke `get_quote` using the DII:

```
Stock::Quoter_var quoter_ref = // get Quoter reference

CORBA::Request_var request =
  quoter_ref->_request ("get_quote");
request->add_in_arg () <<= "IONAY";
request->set_return_type (CORBA::_tc_long);
request->invoke ();
CORBA::Long value;
if (request->return_value () >>= value)
  cout << "Current value of IONA stock: "
       << value << endl;
```

We first obtain a `Quoter` object reference, just as in the SII case. Because all objects support the ability to create `CORBA::Request` instances, we use this reference to create a `CORBA::Request` object. We pass the name of the operation we want to invoke, *i.e.*, `get_quote`, to the `Request` creation operation, and it returns an object reference for the new `Request`.

After creating the `Request`, we fill in the operation arguments. We add an input argument to the `Request` object by first invoking `Request::add_in_arg`, which returns a reference to the `CORBA::Any` that will hold the value of the new argument. We then use the overloaded `operator<<=` to insert the string "IONAY" (the NASDAQ stock symbol for IONA Technologies) into the `Any` as the value of the argument. Next, we set the return type using a `TypeCode` constant for the IDL `long` type.

After all the operation arguments are initialized we invoke the operation.[2] When the `invoke` call returns, we access the return value `Any` owned by the `Request` object, and then use the overloaded `operator>>=` to extract our `CORBA::Long` return value. Finally, we print the return value to the standard output.

This example performs a synchronous twoway invocation. To perform a deferred synchronous invocation, which cannot be done with the SII, we need to replace the `invoke` call, as follows:

```
// Create the request and initialize it as before.

// Send the request.
request->send_deferred ();

// ... continue processing, and then sometime later,
// get the response...
request->get_response ();

// Handle response as before.
```

Note that our call to `get_response` will block if the response is not available immediately. If we want to avoid blocking altogether, we can use `poll_response` instead:

```
// Create the request, then initialize and send
// it as before.
request->send_deferred ();

// Now see if the response has come back.
while (!request->poll_response ()) {
  // ... continue processing ...
}

// When the while loop exits, we are
// guaranteed that the response is available.
// Thus, the following call will not block.
request->get_response ();
// Handle response as before.
```

The `poll_response` is a local invocation on the `Request` that checks to see if the local ORB has received the response from the target object. Once it returns true, we use `get_response` to collect the response. This call is guaranteed not to block once `poll_response` returns true.

Evaluating the DII: As you saw in the example above, the DII requires programmers to write much more code than the SII. In particular, the DII-based application must build the request incrementally and then explicitly ask the ORB to send it to the target object. In contrast, all of the code needed to build and invoke requests with the SII is contained in the generated stubs and hidden from the application.

The increased amount of code required to invoke an operation via the DII yields larger programs that are hard to write and hard to maintain. Moreover, the SII is type-safe because the C++ compiler ensures the correct arguments are passed to the static stubs. Conversely, the DII is not type-safe. Thus, you must make sure to insert the right types into each `Any` or the operation invocation will not succeed.[3]

Of course, if you can't afford to block waiting for responses on twoway calls, you need to decouple the send and receive operations. Historically, in CORBA this meant you were stuck using the DII. A key benefit of the new CORBA Messaging specification is that it effectively allows deferred synchronous calls using static stubs, which alleviates much of the tedium associated with using the DII.

3.2 Oneway Woes

Oneways are used to achieve "fire and forget" semantics while taking advantage of CORBA's typechecking, marshal-

[2] To keep the example from being even more cluttered than it already is, we omitted the exception initialization and don't explicitly test whether the invocation worked.

[3] The DII also allows applications to handle types that were unknown to them at compile time, but in this column we are using the DII only for its deferred synchronous invocation capabilities, not to handle unknown object types.

ing/demarshaling, and operation demultiplexing features. Oneways can be problematic, however, since application developers are responsible for ensuring end-to-end reliability.

Oneway example: To illustrate the challenge of using oneway operations reliably, let's reconsider the IDL interface for the callback handler we defined in [6]:

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

  // Distributed callback information.
  module Callback {
    struct Info {
      string stock_name;
      long value;
    };

    // Distributed callback interface
    // (invoked by the Supplier).
    interface Handler {
      void push (in Info data);
    };

    interface Handler_Registration {
      void unregister ();
    };
  };

  // This is the same as shown above.
  interface Quoter {
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
  };

  interface Notifying_Quoter : Quoter {
    // Register a distributed callback
    // handler that is invoked when the
    // given stock reaches the desired
    // threshold value.
    CallBack::Handler_Registration
      register_callback
           (in string stock_name,
            in long threshold_value,
            in Callback::Handler handler)
      raises (Invalid_Stock);
  };
};
```

Note that our original `Quoter` interface polled servers for stock values. In contrast, the `Callback::Handler` and `Notifying_Quoter` interfaces allow us to receive callbacks when a stock reaches a target threshold value. We create an object of type `Callback::Handler` and pass it to `Notifying_Quoter::register_callback`, along with the name of the stock and its target value. The `register_callback` operation returns an object reference of type `Callback::Handler_Registration`, which allows us to unregister our `Handler` at any time.

When the `Notifying_Quoter` detects that the stock has reached the target value, it invokes the twoway `push` operation on our `Callback::Handler`. While this approach seems simple, it is fraught with subtle and pernicious problems. For example, if our `Handler` is slow to respond, the `Notifying_Quoter` thread that invoked `push` will be blocked. Likewise, if our `Handler` is unreachable due to network congestion or partitioning, the `Notifying_Quoter` thread could be blocked for a lengthy period of time waiting to contact it.

The problems described above may seem minor, but if our `Notifying_Quoter` is busy with thousands of callbacks that are all this problematic, its scalability will be significantly limited. In fact, even one `Handler` could cause a single-threaded `Notifying_Quoter` to hang if its underlying TCP connection becomes flow-controlled.

To avoid these response-related issues for our callbacks, we might be tempted to instead declare the `Handler::push` operation using the `oneway` IDL keyword, as follows:

```
module Stock {
  // ...same as before...

  module Callback {
    // ...same as before...

    interface Handler {
      oneway void push (in Info data);
    };
  };

  // ...
};
```

By declaring `push` as `oneway`, we inform the ORB that the `Notifying_Quoter` should use "fire and forget" semantics when delivering our stock value notifications.

Evaluating oneways: Unfortunately, using `oneway` like this may or may not have the desired effect, for the following reasons:

- **Transport dependencies:** When a oneway is sent using IIOP, there is no response as far as the IIOP subsystem of the requesting ORB is concerned. However, despite the fact that `oneway` is an IDL construct, its implementation depends on the underlying transport used to deliver the request. For instance, IIOP is implemented over TCP, which provides reliable delivery and end-to-end flow control [7]. At the transport level, these features collaborate to suspend a client thread as long as TCP buffers on its associated server are full. Thus, oneways over IIOP are not guaranteed to be non-blocking.

  The situation is even worse for the DCE Common Inter-ORB Protocol (DCE-CIOP), which is the OMG standard DCE-based protocol [8]. DCE-CIOP uses fully synchronous DCE RPC calls to deliver oneway invocations because there is no equivalent to oneway semantics in DCE.

- **Best-effort semantics:** CORBA states that oneway operations have "best-effort" semantics, which means that an ORB need not guarantee their delivery. For example, an ORB that supports a proprietary UDP-based transport can send oneway calls over UDP without bothering to check if they arrive. Moreover, an ORB that conforms to the CORBA specification could even toss all oneway requests into the bit bucket and not attempt to deliver them at all![4]. Thus, if you need end-to-end delivery guarantees for your oneway requests, you cannot portably rely on oneway semantics.

[4] It is unlikely that such an ORB would be well-received in the marketplace, of course!

So, as you can see from the discussion above, oneway just doesn't cut it for most applications. Oneways guarantee neither non-blocking semantics nor reliable delivery. Thus, moving from ORB to ORB you can never be sure of what semantics you'll get.

Until recently, the CORBA communication model didn't provide a standard solution to the limitations described above. Fortunately, the situation is now much improved with the new CORBA Messaging specification [1]. In our next column, we'll show how this specification fixes the limitations with the tedium of programming with deferred synchronous operations via the DII and the weak semantics of oneway operations. For now, however, we'll introduce you to the CORBA Messaging specification.

4 An Introduction to CORBA Messaging

The new CORBA Messaging specification introduces several important features to CORBA. This section presents an overview of the three most important features: asynchronous method invocation (AMI), time-independent invocation (TII), and messaging quality of service (QoS) policies.

4.1 Asynchronous Method Invocation (AMI)

If you read Section 2 carefully, you'll note that standard CORBA doesn't define a truly asynchronous method invocation model using the SII. A common workaround for the lack of asynchronous operations is to use separate threads for each twoway operation. However, the complexity of threads makes it hard to develop portable, efficient, and scalable multi-threaded distributed applications [9]. Moreover, since support for multi-threading is inadequately defined in the CORBA specification there is significant diversity among ORB implementations [10].

Another common workaround to simulate asynchronous behavior in CORBA is to use oneway operations. For instance, a client can invoke a oneway operation to a target object and pass along an object reference to itself. The target object on the server can then use this object reference to invoke another oneway operation back on the original client. However, this design incurs all the reliability problems with oneway operations described in Section 3.2.

To address these issues, CORBA Messaging defines an asynchronous method invocation (AMI) specification that supports the following two models:

Polling model: In this model, the asynchronous twoway invocation returns a `Poller` *valuetype*, which is a new IDL type introduced by the new *Objects-by-Value* specification [11]. A `valuetype` is very much like a C++ or Java class, in that it has both methods and data members.

The client can use the `Poller` methods, which are just local C++ function calls and not distributed invocations, to obtain the status of the request and the value of the reply from the server. If the reply hasn't returned from the server yet, the client can elect to block awaiting its arrival, just like with the deferred synchronous mode described in Section 2. Likewise, it can simply return to the calling thread immediately and check back later on the `Poller`.

Callback model: In this model, an object reference to a object called a `ReplyHandler` is passed as a parameter when a client invokes a twoway asynchronous operation on a server. When the server responds, the client ORB receives the response and invokes the appropriate C++ method on the `ReplyHandler` callback object to handle the reply. The callback model is particularly useful since it relieves the client from having to poll for the result.

An important consequence of both AMI models is that no additional application threads are required in the client. Thus, an application can manage multiple twoway operations simultaneously by using the same thread of control to make overlapping remote requests on one or more objects.

We can use either the polling or callback model to avoid the problems with the DII and oneway approaches described in Section 3. Remarkably, adding asynchrony to the client generally does not require any modifications to the server since the CORBA Messaging specification treats asynchronous invocations as a client-side language mapping issue.

4.2 Time-Independent Invocations (TII)

Time-independent invocations (TII) is a specialization of AMI that supports "store-and-forward" semantics. Time-independent requests may actually outlive the requesting client process, meaning that the response may be gathered by a completely different client. This is useful for applications like email, which require guaranteed delivery of requests to target objects that may not be connected to a network at the time a message is sent. TII is also useful for applications running on "occasionally-connected" clients like laptop computers. By using TII, requests and replies can be delivered to their targets when network connections, routing agents, and QoS properties permit.

To support TII, the CORBA Messaging specification defines a standard *Interoperable Routing Protocol* (IRP) based on the General Inter-ORB Protocol (GIOP).[5] The IRP provides a standard way for time-independent requests to travel between store-and-forward routers built by different vendors. Many solid, stable, production-quality distributed systems in use today are built using asynchronous messaging technology, often referred to as *message-oriented middleware* (MOM) [12]. Thus, IRP also allows ORBs to use these existing MOM products to deliver messages with varying qualities of service (see below), rather than trying to reinvent the wheel.

[5] GIOP is the "abstract base" protocol for building inter-ORB communications, while the Internet Inter-ORB Protocol (IIOP) is simply GIOP implemented over TCP.

4.3 Messaging Quality of Service (QoS)

The CORBA 2.2 specification does not define key QoS features associated with MOM products, such as the TIBCO Information Bus and IBM's MQSeries. Common QoS features include delivery quality, queue management, and message priority.

One of the strengths of the new CORBA Messaging specification is its uniform QoS framework that supports both asynchronous and synchronous method invocations. In this framework, all QoS properties are defined as interfaces derived from `CORBA::Policy`, which we described in our recent columns on the POA [13, 14, 4, 5]. This QoS framework allows applications to define QoS properties at multiple client-side levels:

1. ORB level: QoS policies for the ORB control the QoS for all requests made using that ORB.

2. Thread level: QoS policies may be set on a per-thread basis to control all requests issued from a given thread. Each thread-level policy setting overrides the corresponding setting at the ORB level.

3. Object reference level: QoS policies may be set on each object reference to control requests made using that object reference. Each policy setting at the object reference level overrides the corresponding setting at both the thread level and the ORB level.

Client-side policies allow applications to control details of request and reply delivery if they so choose. For example, they provide control over request and reply timeouts, priorities and ordering, rebinding to servers, and required routing semantics, *i.e.*, whether store-and-forward delivery is needed. Existing CORBA applications that just use the ORB "as-is" need not change, however – the defaults for these request delivery issues provide behavior that is identical to existing CORBA request delivery semantics.

In addition, there are server-side policy management interfaces that allow applications to set desired QoS levels for their objects. These policies are set on a POA as it is created; any objects created under that POA have those policies embedded in their object references. For example, all the objects created under a certain POA might require that they only be invoked within the context of a transaction. By setting the right `TransactionPolicy` on the POA when it's created, each object reference you create within that POA will hold information about the transactional requirements of the object it refers to. Client ORBs can use that information to reject requests made outside of the appropriate transactional context.

In general, applications can specify their QoS requirements to the ORB in a portable, protocol independent, and convenient way. For instance, applications can use the Messaging QoS framework to guide implicit protocol selection by the ORB.

5 Conclusion

Asynchronous messaging is an extremely useful tool to have in your distributed application development toolkit. However, until recently, the CORBA specification did not support this feature. This column illustrated the limitations with common workarounds, such as oneway operations and using DII for deferred synchronous operations, for CORBA's current lack of asynchronous messaging. We then outlined how the new CORBA Messaging specification addresses these limitations. Support for asynchronous messaging will allow CORBA to support a much broader range of application domains.

Our next column will explain in detail how to use the Callback and Polling asynchronous invocation models. In keeping with tradition, we will show many C++ code examples that illustrate these new features.

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`.

Acknowledgements

Thanks to Michi Henning <michi@dstc.edu.au> for providing comments on this column.

References

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

[2] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[3] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object," *C++ Report*, vol. 8, July 1996.

[4] D. C. Schmidt and S. Vinoski, "Developing C++ Servant Classes Using the Portable Object Adapter," *C++ Report*, vol. 10, June 1998.

[5] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter," *C++ Report*, vol. 10, Sept. 1998.

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

[7] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[8] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[9] J. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)," in *USENIX Winter Technical Conference*, (San Diego, CA), USENIX, Jan. 1996.

[10] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

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

[12] S. Maffeis and D. C. Schmidt, "Constructing Reliable Distributed Communication Systems with CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[13] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology," *C++ Report*, vol. 9, November/December 1997.

[14] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, vol. 10, April 1998.