Object Interconnections

An Overview of the OMG CORBA Messaging
Quality of Service (QoS) Framework
(Column 19)

Douglas C. Schmidt
schmidt@cs.wustl.edu
Electrical & Computer Engineering Dept.
University of California, Irvine, CA 92697

Steve Vinoski
vinoski@iona.com
IONA Technologies, Inc.
200 West St., Waltham, MA 02154

This column will appear in the March 2000 issue of the C++ Report magazine.

1 Introduction

This is the final column in our series covering the OMG CORBA Messaging specification [1]. Our previous columns in this series covered the communications models supplied by Messaging [2], explained how to program asynchronous method invocations (AMI) in C++ [3], and described time-independent invocation (TII) and interoperable routing [4]. We finish this series by highlighting the quality of service (QoS) framework supplied by the OMG Messaging specification.

Quality of service (QoS) is a widely accepted term that describes activities and technologies designed to improve and control communication-oriented resource management for applications and systems [5]. Many distributed applications need to selectively configure and optimize various QoS aspects, such as the end-to-end latency of particular requests, the aggregate throughput over some interval, the reliability of one-way message delivery, or how long a client spends waiting for a reply before it times out. Therefore, the OMG Messaging specification defines a QoS framework that allows applications to configure and control various aspects of ORB behavior. This framework defines a set of policy objects, a framework for managing the policies, and extensions to GIOP/IIOP that communicate policies between ORBs.

2 An Overview of QoS Policies from the OMG Messaging Specification

In this section we describe the key QoS features defined in the OMG Messaging specification. We focus on the policies that are available to applications and explain where and how applications can use these policies.

2.1 Client and Server Policy Management Levels

Client policies: Figure 1 illustrates the four levels at which

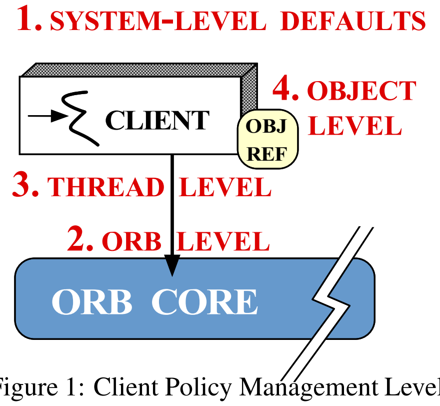

Figure 1: Client Policy Management Levels

a CORBA client application can establish policies. Below, we outline each of these policy levels, ranging from the most coarse-grained control to the most fine-grained control.

1. System-level defaults: ORB implementations supply a set of system-wide policy defaults that apply if they are not overridden at any of the levels described below. Because system-level QoS policy defaults are not standardized, portable applications concerned with QoS must be sure to override the desired QoS policies at the appropriate level.

2. ORB-level policies: By setting policies on an ORB instance, the client can override system-level defaults to control the QoS for all requests made through that ORB. When a client obtains an object reference, that reference is associated with the ORB instance used to make the request. Object references can be obtained either through `ORB::resolve_initial_references`, through `ORB::string_to_object`, or by receiving the object reference from another operation invocation.

Because most CORBA applications use only one ORB instance, you may be surprised to learn that an application can use multiple ORBs simultaneously.[1] However, this should not surprise you given that the ORB itself is a (locality-constrained) object with an IDL interface. Therefore, just as applications can create multiple instances of any CORBA

[1] The phrase "multiple ORBs" often evokes images of linking two different ORB implementations together in the same application, but that's an altogether different issue.

object, they can create multiple instances of an ORB by invoking `CORBA::ORB_init` multiple times with different arguments.

Prior to the OMG's adoption of the Messaging specification, applications rarely had any need to instantiate multiple ORBs. Now that QoS policies can be applied at the ORB level, however, applications may want to use multiple ORBs so they can apply different QoS policies to different invocations. For example, an application might want to communicate with one set of objects using traditional synchronous invocations, but communicate with another set of objects using asynchronous invocations. Such an application might opt to use two different ORBs, with different QoS policies, to communicate with these two sets of objects.

3. Thread-level policies: Applications can override ORB-level and system-level policies on a per-thread basis. This provides a finer granularity of control that allows requests made in a given thread to have different QoS characteristics than requests made by other threads in the same application process. Applications can override thread-level policies by retrieving a `PolicyCurrent` object from the ORB's `resolve_initial_references` bootstrapping operation and invoking appropriate QoS framework operations, such as `set_policy_overrides`.

4. Object-level policies: The finest level of QoS granularity control available to applications is on a per-object reference basis. The `CORBA::Object` interface supplies operations to override thread-, ORB-, and system-level QoS policies, as well as to query the *effective* client-side policy in effect for a given policy type. The ORB implementation computes the effective policy value by considering the system-level default and applying any overrides from the ORB-, thread-, and object-levels.

Server policies: Server applications can provide QoS policies along with the normal POA policies when they call the `create_POA` operation on the `PortableServer::POA` class. When an object reference is created using a QoS-enabled POA, the POA ensures that any server-side policies that affect client-side requests, such as a request priority policy, are embedded in the TAG_POLICIES component in the object reference. This enables clients who invoke operations on such object references to honor the policies required by the target object. Note that not all policies are usable on both clients and servers – see the OMG Messaging specification [1] for more details.

2.2 Policy Types

The standard QoS policies defined in the OMG Messaging specification provide substantial power and flexibility for CORBA applications. Below, we briefly outline some of the standard QoS policy types. Section 3 then describes several common policies in more detail and illustrates how to program them in C++.

Rebind policy: Traditional CORBA applications had no standard way to control whether and how the client ORB transparently rebinds if a LOCATION FORWARD response is received or if a connection drops. The rebind policy allows applications to select whether to transparently rebind (which is the traditional CORBA behavior), rebind when connections are closed but not rebind on LOCATION FORWARD responses, or not rebind at all. In addition, the Messaging specification adds a `CORBA::Object::validate_connection` operation that allows applications to control rebinding explicitly, regardless of the rebind policy in effect.

Synchronization policy: This policy allows applications to control the semantics of *one-way* operations explicitly. Four categories of one-way invocations are defined, each providing different levels of reliability, features, and latency overhead. Section 3.2 explores this policy in more detail and shows how to program it in C++.

Request and reply priority: A time-independent invocation (TII) router [4] accepts CORBA requests and replies and stores them persistently until it can forward them to the next router. The request and reply policies are used in conjunction with TII routers to determine the order in which requests and replies are stored and forwarded at routers. Higher priority requests and replies are given preferential treatment in router queues.

Request and reply timeout: These policies allow applications to control several time-related aspects of request and reply delivery. For example, an application can specify the time window during which the ORB is allowed to deliver a given request or reply to its target. Likewise, an client can control the relative time during which the ORB may deliver a request. In addition, a client can control the round-trip time allowed for a request and its reply. After the specified time has elapsed, the ORB raises a CORBA::TIMEOUT exception. The application can then perform whatever action is necessary to handle the potential failure. Section 3.1 explores this policy in more detail and shows how to program it in C++.

Request routing: An application can specify that a request should not be routed but should be delivered directly by the client ORB. Conversely, an application can specify that the request be sent to the target through a particular router. If an application chooses to use a router, it can specify either asynchronous method invocation [3] or time-independent invocation [4] (*i.e.*, store-and-forward semantics). Another policy, *MaxHops*, allows applications to control the maximum number of hops a request may take when traversing routers on its way from the client to the target.

Queue order: This policy can control queueing behavior for requests that are sent through routers. For instance, applications can specify any order, temporal order, priority order, or ordering based on request deadlines [6].

Unfortunately, we do not have enough column space to provide more details on all of these policies. For more information, please refer to the OMG CORBA Messaging specification.

3 QoS Framework Programming Examples

As shown in Section 2.2, there are a number of policies defined in the OMG Messaging specification. Many of these policies are intended for use with the TII and interoperable routing mechanisms, which makes them somewhat esoteric because there aren't any widely available implementations of these mechanisms (yet). Other policies are useful with conventional CORBA applications, however. In this section we describe how to program two particularly useful policies: *client timeouts* and *reliable one-ways*.

3.1 Client Timeouts

Overview: Many client applications must bound the amount of time they block waiting for the ORB and the server to process a request end-to-end. This feature is particularly important for fault tolerance, *e.g.*, by detecting unresponsive or blocked servers and taking appropriate action, such as migrating work to alternative servers or reporting a possible system failure that requires automatic or manual repair. In all these cases, it is desirable to time out operation invocations after an expected execution time has elapsed.

To support these use-cases, the OMG Messaging specification includes policies that client applications can set to automatically timeout operation invocations. The OMG Messaging specification defines a total of five different timeout policies. We focus our discussion here on the *RelativeRoundtripTimeout* policy, which is the most useful for conventional CORBA applications. This policy affects only the behavior of clients, *i.e.*, no timing information is passed to the server.

At the start of each operation invocation, the client ORB queries the `RelativeRoundtripTimeoutPolicy` defined in the `Messaging` interface. If this policy has been set by the application, the associated timeout is used to bound the amount of time spent (1) establishing the connection to the remote server, (2) sending the request to the server, and (3) waiting for the reply. Before the ORB waits on any potential blocking operation, it computes the time elapsed since the start of the request and resets the timeout value to the remaining time specified by the policy. Thus, if the total timeout value is 15 msecs and the ORB takes 5 msecs to establish the connection, only 10 msecs are available to send the request and receive the reply.

If the timeout expires during any point in the processing sequence, the client ORB reclaims the resources used for the outstanding request and raises the CORBA::TIMEOUT exception. Due to network delays, or simply because the server takes more than the expected time to process a request, it is possible for a reply to arrive *after* a client ORB has raised the TIMEOUT exception. Thus, an ORB must be prepared to receive and ignore replies for requests that the client no longer cares about.

Programming client timeouts: The following example illustrates how to program client timeouts using the *RelativeRoundtripTimeout* policy. As usual, this example is based on the following stock `Quoter` interface, which was first introduced in [7]:

```
module Stock {
  exception Invalid_Stock {};

  interface Quoter {
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
  };
};
```

We'll start our example by defining the following helper function:

```
void
timed_quote (Quoter_ptr quoter)
{
  try {
    const char *stockname = "ACME ORB inc."

    CORBA::Long value =
      quoter->get_quote (stockname);

    // Print the stock value.
    cout << "stock " << stockname
         << " = " << value << endl;
  } catch (CORBA::TIMEOUT &timeout) {
    // Handle timeout exception...
  } catch (Stock::Invalid_Stock&) {
    // Handle Invalid_Stock exception...
  } catch (...) {
    // Handle other exceptions.
  }
}
```

This function tries to get and print the current value of `stockname` by invoking the `get_quote` operation via the `quoter` object reference. The `get_quote` method is invoked under timeout control, based on the QoS override techniques shown below. If a CORBA::TIMEOUT or some other exception is thrown, the function takes corrective action.

We implement our `main` driver function next. As usual, we perform general initialization activities first, starting by obtaining object references to an ORB and two locality constrained objects, `PolicyManager` and `PolicyCurrent`, defined by the OMG Messaging specification.

```
int main (int argc, char *argv[])
{
  try {
    CORBA::ORB_var orb =
      CORBA::ORB_init (argc, argv);

    CORBA::Object_var object =
      orb->resolve_initial_references
      ("ORBPolicyManager");

    CORBA::PolicyManager_var policy_manager =
      CORBA::PolicyManager::_narrow (object.in ());
```

```
    object = orb->resolve_initial_references
      ("PolicyCurrent");

    CORBA::PolicyCurrent_var policy_current =
      CORBA::PolicyCurrent::_narrow (object.in ());
```

The `PolicyManager` object is used to override ORB-level QoS policies, whereas the `PolicyCurrent` object is used to override thread-level QoS policies.

As we pointed out in Section 2.1, the OMG Messaging specification does not define default system-level QoS policies. Therefore, we must first initialize the QoS policy defaults in the `PolicyManager` and `PolicyCurrent` to be "no-ops." This will allow us to override them selectively later on without worrying about side-effects from undesired defaults.

```
    CORBA::PolicyList policy_list;

    // Disable all default policies.
    policy_list.length (0);
    policy_manager->set_policy_overrides
      (policy_list,
       CORBA::SET_OVERRIDE);
    policy_current->set_policy_overrides
      (policy_list,
       CORBA::SET_OVERRIDE);
```

Now that we've finished general ORB and Messaging initialization, we can perform application-specific initialization. First, we'll obtain an object reference to a `Quoter`.

```
    const char *IOR;
    // ... assume the IOR is initialized somehow.

    object = orb->string_to_object (IOR);

    Quoter_var quoter =
      Quoter::_narrow (object.in ());
```

For simplicity, we've "hard-coded" the IOR into the program. A more sophisticated client would obtain the `Quoter`'s object reference using a factory or a standard CORBA Object Service, such as Naming or Trading.

Next, we'll create a 1 second timeout for each policy level, *i.e.*, ORB-level, thread-level, and object-level, as follows:

```
    // 1 second (TimeT has 100 nanosecond resolution).
    TimeBase::TimeT timeout = 10000000;

    CORBA::Any orb_timeout;
    CORBA::Any thread_timeout;
    CORBA::Any object_timeout;

    orb_timeout <<= timeout;
    thread_timeout <<= timeout;
    object_timeout <<= timeout;
```

Note how the timeout values are stored in CORBA `Any`s because that's the type expected by the `ORB`'s `create_policy` operation's second parameter.

Now that we've created the timeouts, we'll use them to set the *RelativeRoundtripTimeout* policy at various levels. We'll first override the ORB-level policy and invoke a timed `get_quote` call, as follows:

```
    // Override the ORB policies.
    policy_list.length (1);
    policy_list[0] = orb->create_policy
      (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE,
       orb_timeout);

    policy_manager->set_policy_overrides
      (policy_list, CORBA::SET_OVERRIDE);

    // Invoke the get_quote() operator.
    timed_quote (quoter.in ());

    // Cleanup.
    policy_list[0]->destroy ();
```

After we override the ORB's *RelativeRoundtripTimeout* policy, any other invocations that use this ORB will have implicitly have a 1 second timeout. Therefore, applications should be careful when overriding ORB-level QoS policies because unexpected side-effects can occur.

One way to minimize side-effects is to limit the level at which QoS policies are overridden. For instance, we can override the *RelativeRoundtripTimeout* policy at the thread-level and invoke another timed `get_quote` call, as follows:

```
    // Override the thread policies.
    policy_list.length (1);
    policy_list[0] = orb->create_policy
      (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE,
       thread_timeout);

    policy_current->set_policy_overrides
      (policy_list,
       CORBA::SET_OVERRIDE);

    // Invoke the get_quote() operator.
    timed_quote (quoter.in ());

    // Cleanup.
    policy_list[0]->destroy ();
```

Unlike the ORB-level override shown earlier, this thread-level override only affects the current thread.

Even overriding at the thread-level may have undesirable side-effects, however. Therefore, our final example illustrates how to selectively override the *RelativeRoundtripTimeout* policy at the object-level, which is the finest level of granularity supported by the OMG Messaging QoS policy framework.

```
    // Override the object policies.
    policy_list.length (1);
    policy_list[0] = orb->create_policy
     (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE,
      object_timeout);

    // Create a new object reference!
    object = quoter->_set_policy_overrides
      (policy_list, CORBA::SET_OVERRIDE);

    Quoter_var timed_quoter =
      Quoter::_narrow (object.in ());

    // Invoke the get_quote() operator.
    timed_quote (timed_quoter.in ());

    // Cleanup.
    policy_list[0]->destroy ();
  } catch (...) {
    // Handle exceptions...
  }
}
```

We invoke the _set_policy_overrides operation on the quoter object reference itself to create a timed_quoter object reference, which is imbued with the designated *RelativeRoundtripTimeout* policy.[2] This new object reference is used to invoke the timed get_quote operation. Due to the "immutability of object references" [1], the original quoter object reference is not altered in any way.

3.2 Reliable One-ways

Overview: Traditional CORBA one-way semantics are often unacceptable because there is no guarantee that a particular invocation will be delivered [2]. Moreover, location forwarding does not occur with traditional one-way invocations, which can complicate load balancing, fault tolerance, and automatic server activation [8].

To address limitations with one-way invocations in the earlier CORBA standard, therefore, the OMG Messaging specification contains a *SyncScope* policy. This policy is set by a client and uses new flags in the response_requested field of the GIOP header. The server ORB checks this field to determine what type of a reply, if any, is required for a one-way invocation. Figure 2 illustrates where each of the following four synchronization

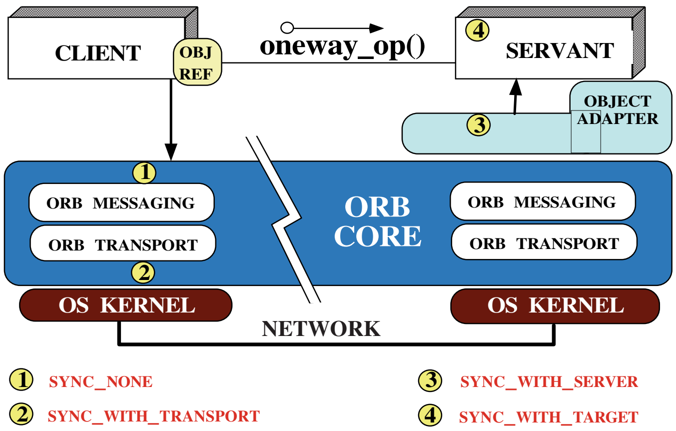

Figure 2: Reliable One-way Synchronization Scopes

scopes are defined in the OMG Messaging specification.

1. SYNC_NONE: With this option, the client ORB returns control to the client application before passing the request to the transport layer. Although the client will not block, there is no location forwarding or acknowledgement of reception. This policy is useful for applications that can tolerate some degree of request lossage.

2. SYNC_WITH_TRANSPORT: With this option, the ORB returns control to the client only after the request is passed successfully to the transport layer, *e.g.*, the client-side TCP protocol stack. In this case, the client can block if local resources are unavailable. However, there is still no location forwarding or acknowledgement of reception.

3. SYNC_WITH_SERVER: With this option, the server sends a reply after invoking any servant managers, but before it dispatches the request to the target object. A reply of NO_EXCEPTION implies that all location forwarding has been performed and the client ORB can return control to the client application. Thus, the client will block only as long as it takes for the invocation to be processed by the ORB Core and the remote Object Adapter. The SYNC_WITH_SERVER option provides a client with an assurance that the remote servant has been located. This feature is particularly useful for real-time applications that require some degree of reliability, but do not need to wait for the ultimate upcall to complete.

4. SYNC_WITH_TARGET: This option is equivalent to a synchronous two-way CORBA operation, *i.e.*, the client will block until the server ORB sends a reply after the target object has processed the operation. Any location forwarding will have occurred and a SYSTEM_EXCEPTION reply can be sent if problems occur. If no exception is raised, the client can assume that the target servant processed its request.

With the availability of the *SyncScope* policy, the oneway keyword is now redundant. In particular, you can convert a two-way operation with no return value or inout or out parameters into a one-way by defining an appropriate policy and using the CORBA Messaging QoS policy framework. Even so, the oneway keyword remains in OMG IDL for backward compatibility.

Programming reliable one-ways: To illustrate the use of the *SyncScope* policy and reliable oneway operations, let's revisit the IDL interface for the callback handler we defined in [2]:

```
module Stock {
  module Callback {
    struct Info {
      string stock_name;
      long value;
    };

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

  // ...
```

The Handler interface defines a one-way push method that is used to pass a stock name and its associated value from a supplier to a consumer that has subscribed to receive callbacks. Because push returns no information to its supplier, we define it as a one-way operation. In the example below, we illustrate how to program a supplier application using various *SyncScope* policies outlined earlier.

As usual, we start by initializing the ORB and policy-related objects, and then obtain an object reference to a Callback::Handler, as follows:

[2] Note that the _set_policy_overrides operation supplied by CORBA::Object has a leading underscore so that it, like other CORBA::Object member functions, does not clash with user-defined operation names defined in derived interfaces.

```
int main (int argc, char *argv[])
{
  // Initialization activities...
  CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv);
  object = orb->resolve_initial_references
    ("ORBPolicyManager");
  CORBA::PolicyManager_var policy_manager =
    CORBA::PolicyManager::_narrow (object.in ());
  object = orb->resolve_initial_references
    ("PolicyCurrent");
  CORBA::PolicyCurrent_var policy_current =
    CORBA::PolicyCurrent::_narrow (object.in ());

  const char *IOR;
  // ... assume the IOR is initialized somehow.

  CORBA::Object_var object =
    orb->string_to_object (IOR);

  // Convert object to concrete type.
  Stock::Callback::Handler_var handler =
    Stock::Callback::Handler::_narrow
      (object.in ());

  CORBA::PolicyList policy_list;
  policy_list.length (1);
```

Next, we override the ORB-level *SyncScope* policy to be SYNC_NONE and push the current threshold value of the ubiquitous "ACME ORB, Inc." stock to the callback consumer.

```
  CORBA::Any orb_level;
  orb_level <<= Messaging::SYNC_NONE;
  policy_list[0] = orb->create_policy
    (Messaging::SYNC_SCOPE_POLICY_TYPE,
     orb_level);
  policy_manager->set_policy_overrides
    (policy_list,
     CORBA::SET_OVERRIDE);

  Stock::Callback::Info info;

  // Assign name and value.
  info.stock_name =
    CORBA::string_dup ("ACME ORB Inc.");
  info.threshold_value = TRADING_THRESHOLD;

  // Push this info to the callback consumer.
  handler->push (info);
  policy_list[0]->destroy ();
```

Since the SYNC_NONE policy is effectively a "no-op," this example will have no better reliability semantics than a conventional one-way operation. Therefore, to ensure that the server at least receives the one-way operation, we can set the *SyncScope* policy to the SYNC_WITH_SERVER value. For variety, we override this policy at the thread-level, as follows:

```
  CORBA::Any thread_level;
  thread_level <<= Messaging::SYNC_WITH_SERVER;
  policy_list[0] = orb->create_policy
    (Messaging::SYNC_SCOPE_POLICY_TYPE,
     thread_level);
  policy_current->set_policy_overrides
    (policy_list,
     CORBA::SET_OVERRIDE);

  // Push this info to the callback consumer.
  handler->push (info);
  policy_list[0]->destroy ();
```

When the server ORB receives the `push` operation it will send back an acknowledgement just before invoking the upcall on the `Callback::Handler` servant.

Finally, we show how to set the *SyncScope* policy to the SYNC_WITH_TARGET value, which we override at the object-level, as follows:

```
  CORBA::Any object_level;
  object_level <<= Messaging::SYNC_WITH_TARGET;
  policy_list[0] = orb->create_policy
    (Messaging::SYNC_SCOPE_POLICY_TYPE,
     object_level);
  object = handler->_set_policy_overrides
    (policy_list,
     CORBA::SET_OVERRIDE);
  handler =
    Stock::Callback::Handler::_narrow (object.in ());

  // Push this info to the callback consumer.
  handler->push (info);
  policy_list[0]->destroy ();
}
```

Note that the SYNC_WITH_TARGET policy value yields semantics that are roughly equivalent to a two-way operation. The only difference is that a two-way operation can raise user-defined exceptions, whereas a one-way operation cannot.

As you can see, this example is structurally similar to the timeout example from Section 3.1. The main difference is the use of different policies types and values.

4 Concluding Remarks

This column outlines the myriad QoS policies defined in the OMG CORBA Messaging specification. As shown above, these QoS policies provide a new degree of flexibility and control to CORBA application developers. We focused our discussion on several policies, *client timeouts* and *reliable one-ways*, that we've found useful when developing real-world CORBA applications.

When evaluating if and how to use the OMG Messaging QoS policy framework, we recommend that you consider the following points.

Programming complexity: Although the QoS policy framework is very powerful, it comes with an increased cost in programming complexity. This additional complexity is necessary to allow CORBA to expand its role as an integration technology and provide the hooks needed to accommodate message-oriented middleware (MOM) solutions within standards-based CORBA applications. Traditionally, MOM services have supplied rich QoS configurability. Thus, enabling CORBA to interact in the MOM space required it to supply similar QoS capabilities.

Portability: At this point, no ORBs support the entire OMG CORBA Messaging specification, though both Orbix 2000 [9] and TAO [10] support many of the CORBA Messaging features. Therefore, before jumping in head first and using these new features in your next mission-critical

CORBA project, you should check to see which of the QoS policy framework and associated features your vendor supports. For example, some ORB implementations use Singleton ORBs and cannot cope with multiple active ORB instances within a single application process.

Quality: ORB implementations that are not specifically architected and designed to accommodate policy overrides when establishing bindings from clients to targets will be hard pressed to support these features easily, robustly, or efficiently. Therefore, we recommend that you carefully evaluate your ORB to determine how accurately its implementation reflects the OMG CORBA Messaging specification and how well it implements the Messaging features. We can tell you from firsthand experience with TAO [10] and Orbix 2000 [9] that implementing these features efficiently and robustly is far from trivial.

Our next column will go back to basics and talk about the C++ mapping, how it works, and why it is the way it is. In addition, we'll talk about what a "modern" C++ mapping might look like if it dropped compatibility with C and used ISO/ANSI C++ features instead. As always, send us email at `object_connect@cs.wustl.edu` if you have any questions or comments.

Acknowledgements

Thanks to Jeff Parsons for his help constructing the reliable one-way example.

References

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

[2] D. C. Schmidt and S. Vinoski, "Introduction to CORBA Messaging," *C++ Report*, vol. 10, November/December 1998.

[3] D. C. Schmidt and S. Vinoski, "Programming Asynchronous Method Invocations with CORBA Messaging," *C++ Report*, vol. 11, February 1999.

[4] D. C. Schmidt and S. Vinoski, "Time-Independent Invocation and Interoperable Routing," *C++ Report*, vol. 11, April 1999.

[5] C. D. Gill, F. Kuhns, D. L. Levine, D. C. Schmidt, B. S. Doerr, R. E. Schantz, and A. K. Atlas, "Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems," in *Proceedings of the 1st IEEE International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, Nov. 1999.

[6] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, to appear 2000.

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

[8] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[9] IONA Technologies, "Orbix 2000." www.iona-iportal.com/suite/orbix2000.htm.

[10] Center for Distributed Object Computing, "TAO: A High-performance, Real-time Object Request Broker (ORB)." www.cs.wustl.edu/~schmidt/TAO.html, Washington University.