Object Interconnections

Collocation Optimizations for CORBA (Column 18)

Douglas C. Schmidt and Nanbor Wang
{schmidt,nanbor}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

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

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

1 Introduction

In this column, we take break from our ongoing discussion of the CORBA Messaging specification to cover *collocation*, which is an important topic for component-based application developers. Collocation is a technique for transparently optimizing communication overhead when clients and servants are configured into the same address space. For instance, Microsoft COM [1] applications frequently use collocation to access so-called "in-proc" components.

Unlike COM, which evolved from its original collocated component model into a networked component model, CORBA is often considered to be a *distributed* object computing model. However, there are situations where clients and servants must be configured in the same address space [2]. For example, user-supplied `ServantManagers` are ordinary CORBA objects that are invoked by POAs to help incarnate servants [3]. In such cases, CORBA ORBs can transparently employ *collocation optimizations* to ensure there's no unnecessary overhead of (de)marshaling data or transmitting requests/replies through a "loopback" communication device.

In this column, we describe and evaluate several collocation techniques. To make the discussion concrete, we describe how collocation is supported in TAO [4], which is an ORB developed at Washington University, St. Louis. Finally, we present benchmarking results that compare the relative performance gains from various types of collocation strategies and optimizations.

2 Motivating Example

To make our discussion more concrete, consider the following client that uses the standard OMG Naming service to locate and invoke an operation on a Stock `Quoter` object:

```
int main (int argc, char *argv[])
{
  // Initialize the ORB.
  CORBA::ORB_var orb =
    CORBA::ORB_init (argc, argv);
  // Get reference to name service.
  CORBA::Object_var obj =
    orb->resolve_initial_references ("NameService");

  // 1. Opportunity for collocation.
  CosNaming::NamingContext_var name_context =
    CosNaming::NamingContext::_narrow (obj);

  // Create desired service name.
  const char *name = "Quoter";
  CosNaming::Name service_name;
  service_name.length(1);
  service_name[0].id = name;

  // Find object reference in Naming Service.
  Object_var obj =
    name_context->resolve (name);

  // 2. Opportunity for collocation.
  Quoter_var q = Quoter::_narrow (obj);

  const char *stock_name = "ACME ORB Inc.";

  // Invoke call, which may be collocated.
  long value = q->get_quote (stock_name);
  cout << "value of " << stock_name
       << " = $"
       << value << endl;
}
```

As shown above, the client gets an object reference to the stock quote service, asks it for the value of the "ACME ORBs, Inc." stock, and prints out the value if everything works correctly.

What's important to note in this example is that both the Naming Service and the Quoter Service may be collocated in the same address space as the client. When this is case, we'd like the ORB to invoke the `resolve` and `get_quote` operations as directly as possible, rather than incurring all the overhead of (de)marshaling data and transmitting requests/replies through a "loopback" communication device. Moreover, the ORB should be able to determine automatically if the servant is configured into the same address space as the client and perform the collocation optimization transparently.

3 Overview of Alternative Collocation Techniques

In this section we evaluate two general techniques for implementation collocation, which we term the *standard* technique and the *direct* technique.

3.1 Standard Collocation

This strategy uses a so-called "collocation-safe" stub to handle operation invocations on a collocated object. Invoking an operation via a collocation-safe stub ensures the following checks are performed:

1. Applicable client policies are used.
2. The server ORB (which may or may not be the ORB used by the invoking client) has not been shutdown.
3. Interceptors are invoked at the proper interception points.
4. The thread-safety of all ORB and POA operations.
5. The POA managing the servant still exists.
6. The POA Manager [5] of this POA is queried to make sure invocations are allowed on the POA's servants.
7. The servant for the collocated object is still active.
8. The `POA::Current`'s context is set up for this upcall.
9. The POA's policies, *e.g.*, the `ThreadPolicy`, `LifespanPolicy`, and `ServantRetentionPolicy`, are respected.

If after all these checks it is safe to invoke the operation, one implementation technique is to have the stub use the `ServantBase` exported from the server's POA, downcast it to the appropriate concrete servant type, and forward the operation directly to the servant operation. Collocation-safe stubs ensure that the `POA::Current` is restored to its context before the current invocation began, various locks in POA and ORB are released, and the internal states of POA are restored after either a successful or unsuccessful operation invocation.

Because the collocated stubs must access the client ORB (*e.g.*, to determine which client policies are enabled) and the server ORB (*e.g.*, to access the object adapter for myriad of CORBA-compliant operations) the collocation-safe stubs must be able to communicate with both ORBs efficiently. Therein lies the rub, *i.e.*, going through all these steps can incur a non-trivial amount of overhead.

One potential drawback of using standard collocation strategy is that it requires the collocation-safe stubs to maintain knowledge about POA skeleton class names and hierarchies. Thus, if an ORB supports both a BOA and a POA, the IDL compiler must generate different stubs depending on which Object Adapter the application uses. Fortunately, the BOA is now obsolete in the current and future CORBA specifications. Therefore, users should be able to select what type of collocated stubs are generated by their IDL compiler.

3.2 Direct Collocation

To minimize the overhead of the standard collocation strategy outlined above, it is possible to implement collocation to forward all requests directly to the servant class. Thus, the POA is not involved at all. When implemented correctly, the performance of direct collocation should be competitive to that of a virtual method call on the servant operation.

However, although the direct strategy provides the maximum performance for collocated operation invocations, the following problems arise:

ORB lifetime: The ORB servicing the object can be shut down at any point. Invoking the collocated servant's operation directly will still succeed, even if the ORB has been shut down. This behavior is incorrect, however, since client's should receive exceptions if they invoke operations after ORBs have shut down.

Object availability: Depending on the policies used by the POA, an object can be activated and then later deactivated or removed from the POA completely. Since direct invocation does not check for the availability of an object, operation invocations can still "succeed" even after an object has been deactivated or removed from the POA. As with ORB lifetime, allowing this behavior violates the object management model provided by CORBA. Moreover, the results will be nonpredictable, at best, and disastrous, at worst, if an operation is invoked directly after a servant has been unregistered from the POA and deleted.

POA Managers: A POA Manager encapsulates the processing state of the POAs it is associated with. Using operations on the POA Manager, an application can cause requests for those POAs to be queued or discarded. Likewise, a POA can be put in the holding, discarding, or inactive state by its POA Manager. Making operation invocations directly on the servant implementations circumvents these mechanisms and prevents applications from controlling the rate at which incoming requests are dispatched by POA Managers and POAs.

POA's threading policy: To integrate non-thread-safe legacy software into newly developed systems, the POA specification defines a `SINGLE_THREADED` policy. POAs using the `SINGLE_THREADED` policy serialize the dispatching of incoming requests. Implementing collocated object optimization by directly invoking servant operations defeats this policy, which may cause race conditions if more than one thread dispatches upcalls simultaneously.

Interceptors: Interceptors [6] allow application developers to specify additional code that is executed before or after normal operation code. These programmable interception points enable applications to perform security checks, provide debugging traps, maintain audit trails, etc. It is necessary that the ORB run these interceptors regardless of the collocation of the client and the server. Direct collocated calls bypass interceptor invocations, which can break local/remote transparency and cause security violations.

Upcall contexts: The CORBA specification mandates that certain pseudo-objects, such as `POA::Current`, service-specific contexts, `RequestID`, and other `Current`-derived objects be available during an upcall. These pseudo-objects provide the context for the request that's currently being serviced. If a request is invoked directly on a servant, these pseudo-objects will either not exist or will return incorrect information for the current request context.

Location transparency: An object can migrate among ORBs. Remote operation invocations can receive a `LOCATE_FORWARD` reply to any request invocation. When forwarding occurs, the client ORB is responsible for transparently delivering the current request and subsequent requests to the location denoted by the new object reference returned in the `LOCATE_FORWARD` reply. An operation invocation to a collocated object may be forwarded to a remote object. Likewise, a remote operation invocation can be forwarded back to a collocated object. For collocation to work transparently, location forwarding must be supported by the ORB's collocation mechanism.

Servant management: The POA makes a clear distinction between a CORBA object and its servant [5]. Specifically, a single CORBA object may be incarnated by multiple servants over its lifetime. Likewise, a single servant might incarnate multiple CORBA objects simultaneously. Applications can use `ServantManagers` to cause the POA to incarnate objects on-demand when requests arrive for them. In many cases, servants do not exist outside the context of a request invocation, *e.g.*, they can be destroyed by the `ServantManager` as soon as they complete the request. Direct collocation assumes that the lifetime of a servant is the same as the lifetime of the object it incarnates, and further assumes that an object reference to a collocated object can be implemented as a C++ pointer.[1] Because these lifetimes are indeed separate, however, such object references can easily become dangling C++ pointers when the servant is destroyed.

Dynamic object management: Some clients use the Dynamic Invocation Interface (DII). The DII is essentially a generic stub that allows clients to build requests at run-time, without requiring the client to have interface-specific stubs compiled in. Similarly, some servers implement their CORBA objects using the Dynamic Skeleton Interface (DSI). The DSI allows objects to receive requests without requiring the server to have interface-specific skeletons compiled in. Direct collocation assumes that both clients and objects are implemented using static stubs and skeletons, respectively, thus disallowing collocation optimizations in the dynamic case and penalizing DII- and DSI-based applications.

Priority inversion: Collocated operation invocations are run in the client's thread-of-control. Therefore, directly invoking an operation on a collocated object can cause *priority inversions*, which occurs when lower-priority threads block higher-priority threads from executing [7]. Proper mechanisms must be in place to set up and restore the running thread's priority based on both the client and server's priority policies.

Oneway semantics: Oneway calls are supposed to allow the client to proceed without worrying whether the call was actually delivered to the target object and without waiting for the object to carry out the request. Because collocated oneway calls are invoked within the client's thread of control, oneway invocations turn into synchronous invocations, thus blocking the client for the duration of the invocation.

As described above, the direct collocation strategy breaks the CORBA object model in many ways. Fortunately, it is possible to implement collocation in ORBs that can perform relatively well, while still preserving all the semantics of the CORBA object model. The following section discusses various techniques for achieving these qualities.

4 Implementing Collocation – A Case Study

To support collocation optimizations transparently, an ORB must be able to identify an object's location without explicit application intervention. To complicate matters, however, clients can obtain object references in several ways, *e.g.*, from a CORBA Naming Service, a Trading Service, or from a Lifecycle Service generic factory operation. Likewise, clients can use `string_to_object` to convert a stringified interoperable object reference (IOR) into an object reference.

Regardless of how an object reference is obtained, the ORB must create a proxy or stub for the object in the caller's address space. To ensure locality transparency, therefore, an ORB's collocation optimization must automatically determine if an object is collocated with the caller. If it is, the ORB returns a collocated proxy/stub that implements collocation; if it is not,

[1] Most pre-POA ORBs make this assumption and implement object references in this manner.

the ORB returns a proxy/stub that handles remote calls to a distributed object.

To concretely illustrate how collocation works, this section describes how to optimize collocated client/servant configurations using techniques in TAO [4], which is an open-source, real-time ORB developed at Washington University, St. Louis.

4.1 Overview of TAO's Collocation Optimizations

TAO supports both *standard* and *direct* collocation strategies by creating different collocation stub implementations when an object reference is demarshaled according to a user-selectable flag. TAO uses the standard collocation strategy by default. However, the actual collocation strategy used by the TAO ORB can be set by passing the correct argument into `CORBA::ORB_init` via TAO's `-ORBCollocationStrategy Direct/Thru_POA` option.

TAO's two strategies are outlined below:

Thru_POA: In TAO, the standard collocation strategy is called "`Thru_POA`" to reflect the fact that operation invocations using this strategy always go through the POA. The `Thru_POA` strategy is the default collocation strategy in TAO. In this strategy, a collocation-safe stub is used to handle operation invocations on a collocated object. Invoking an operation on this type of collocated stub ensures:

1. The server ORB (which may or may not be the same ORB as the clients') has not been shut down, by querying the servant's ORB which the stub refers to.
2. The thread-safety of all ORB and POA operations by grabbing the necessary locks.
3. The POA managing the servant still exists by looking up the servant through the POA; this operation also ensures that the proper `ServantLocator` or the `ServantActivator` is called if the POA is using such policies.
4. The POA Manager of this POA is queried to make sure upcalls are allowed to be performed on the POA's servants.
5. The servant for the collocated object is still active.
6. The `POA::Current`'s context is setup for this upcall.
7. The POA's threading policy is respected by querying the POA's threading policy.

If it is safe to invoke the operation, the stub uses the `ServantBase` exported from the server's POA, downcasts it to the appropriate concrete servant type, and forwards the operation directly to the servant operation. Collocation-safe stubs ensure that the `POA::Current` is restored to its previous context before the current invocation, and various locks in POA and ORB are released, either after a successful or an unsuccessful operation invocation.

Direct: In this TAO-specific extension, the collocation class forwards all requests directly to the servant class, *i.e.*, the POA is not involved at all. However, this implementation does not support the following standard POA features: (1) the `POA::Current` is not setup, (2) interceptors are bypassed, (3) POA Manager state is ignored, (4) Servant Managers are not consulted, (5) etherealized servants can cause problems, (6) location forwarding is not supported, and (7) the POA's `Thread_Policy` is circumvented. As shown in Figure 4, supporting all these standard features decreases collocation performance. Therefore, TAO provides the `Direct` strategy that is optimized for real-time applications with very stringent latency requirements.

4.2 Implementing TAO's Collocation Optimizations

Figure 1 shows the classes used in TAO to support both *standard* and *direct* collocation strategies. The stub and skeleton

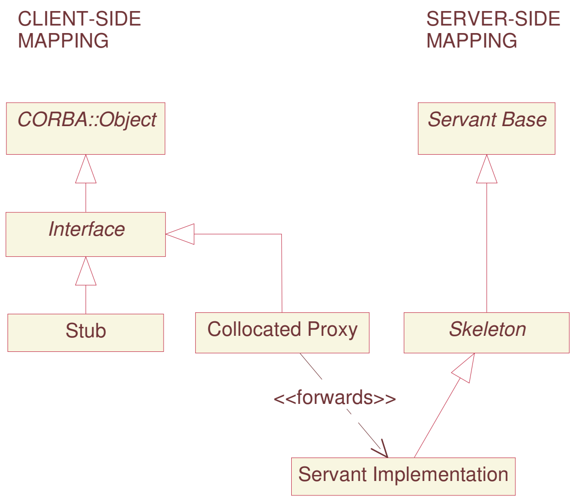

Figure 1: TAO's POA Mapping and Collocation Class

classes shown in Figure 1 are required by the POA specification, though the collocation class is specific to TAO's collocation implementation. Collocation is transparent to the client because the client only knows about the abstract interface and never uses the collocation class directly. As with remote operation invocations, the ORB Core is responsible for locating servants and ensuring that the collocated stub class, rather than the remote stub class, is used by a client when the servant resides in the same address space.

The specific steps used by TAO's collocation optimizations are described below:

Step 1 – Determining collocation: To determine if an object reference is collocated, TAO's ORB Core maintains a *collocation table*. Figure 2 shows the internal structure for collocation table management in TAO. Each collocation table maps

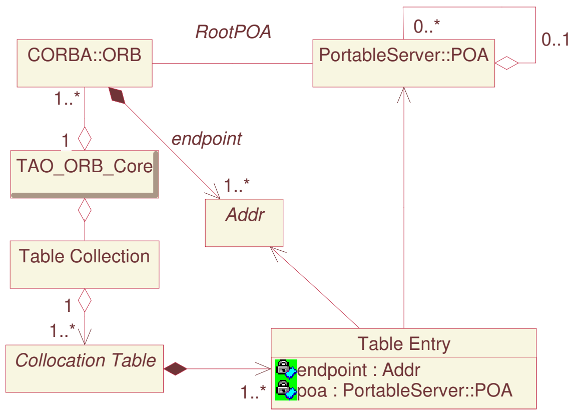

Figure 2: Class Relationship of TAO's Collocation Tables

an ORB's transport endpoints to its RootPOA. In the case of IIOP, for example, endpoints are specified using TCP/IP {port number, host name} tuples.

Multiple ORBs can reside in a single CORBA application process. Each ORB can support multiple transport protocols and accept requests from multiple transport endpoints. Therefore, TAO maintains multiple collocation tables to handle all transport protocols used by ORBs within a single process. Because different protocols have different addressing formats, maintaining protocol-specific collocation tables allows TAO to strategize and optimize the lookup mechanism for each protocol.

Step 2 – Obtaining a reference to a collocated object: A client acquires an object reference either by resolving an imported IOR using `string_to_object` or by demarshaling an incoming object reference. In either case, TAO examines the corresponding collocation tables according to the profiles carried by the object to determine if the object is collocated or not. If the object is collocated, TAO performs the steps shown in Figure 3 to obtain a reference to the collocated object.

When the standard collocation strategy is enabled, the ORB only checks if the imported object reference is collocated or not when it resolves the object reference. To determine this, TAO examines the endpoint information in the collocation table maintained by TAO's ORB Core. If the imported object reference refers to a collocated object, an object reference with

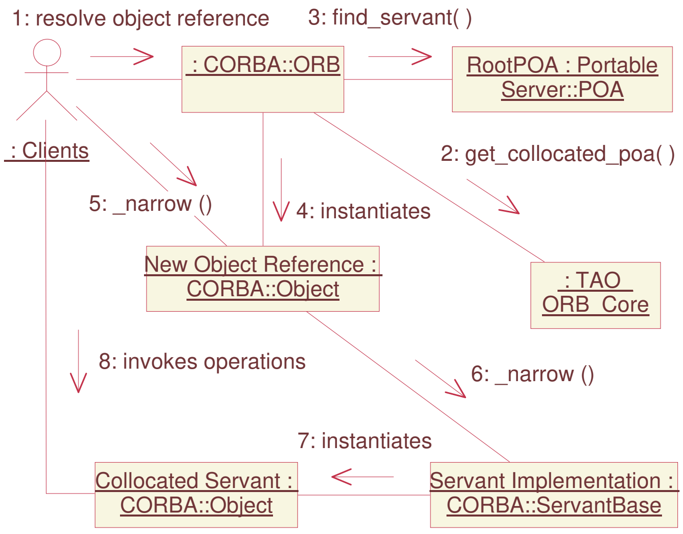

Figure 3: Finding a Collocated Object in TAO

the collocation-safe stub is generated. This stub contains information about the Object Adapter and server ORB associated with the object.

When TAO is configured to use the direct collocation strategy, the ORB resolves an imported object reference using the steps shown in Figure 3. To resolve an object reference **(1)**, the ORB checks **(2)** the collocation table maintained by TAO's ORB Core to determine if any object endpoints are collocated. If a collocated endpoint is found the RootPOA corresponding to the endpoint is returned. Next, the matching Object Adapter is queried for the servant, starting at its RootPOA **(3)**. The ORB then instantiates a generic `CORBA::Object` **(4)** and invokes the `_narrow` operation on it. If a servant is found, the ORB's `_narrow` operation **(5)** invokes the servant's `_narrow` operation **(6)** and a collocated stub is instantiated and returned to the client **(7)**. Finally, clients invoke operations **(8)** on the collocated stub, which forwards the operation to the local servant via a direct virtual method call.

If the imported object reference is not collocated either operation **(2)** or **(3)** will fail. In this case, the ORB invokes the `_is_a` operation to verify that the remote object matches the target type. If the test succeeds, a remote stub is created and returned to the client and all subsequent operations are distributed. Thus, the process of selecting collocated stubs or non-collocated stubs is completely transparent to clients and is performed only when the object reference is created.

Step 3 – Performing collocated object invocations: Collocated operation invocations in TAO borrow the client's thread to execute the servant's operation. Therefore, they are executed within the client thread at its thread priority. Although executing an operation in the client's thread is very efficient, it is undesirable for certain types of real-time applications [8]. For instance, priority inversion can occur when a client in a lower priority thread invokes operations on a collocated object that would otherwise be serviced by a higher priority thread.

Therefore, to provide greater access control over the scope of TAO's collocation optimizations applications can associate different access policies to endpoints so they only appear collocated to certain priority groups. Since endpoints and priority groups in many real-time applications are statically configured, this access control lookup imposes no additional overhead.

5 Performance Results

To measure the performance gain from TAO's collocation optimizations, we ran server and client threads in the same process. The platforms used to benchmark the test program were a quad-CPU 300 Mhz UltraSparc-II running SunOS 5.7 and a dual-CPU 333 Mhz Pentium-II running Microsoft Windows NT 4.0 with SP5. To compare performance systematically, the test program was run with the standard collocation strategy, the direct collocation strategy, as well as with neither collocation optimization, *i.e.*, using remote stubs via the loopback network interface. To compare the performance gain of collocation optimizations to the optimal performance, we also measured the time to perform the exact same task by making direct virtual function calls on the skeleton class.

Figure 4 shows the performance improvement, measured in calls-per-second, using TAO's collocation optimizations. Each operation cubed a variable-length sequence of `longs` that contained 4 and 1,024 elements, respectively. The performance of operation invocations improves dramatically when servants are collocated with clients. With the standard optimization, we obtain a performance improvement of 3,000% to 6,000% comparing to the case when the calls are made thru the local loopback device.

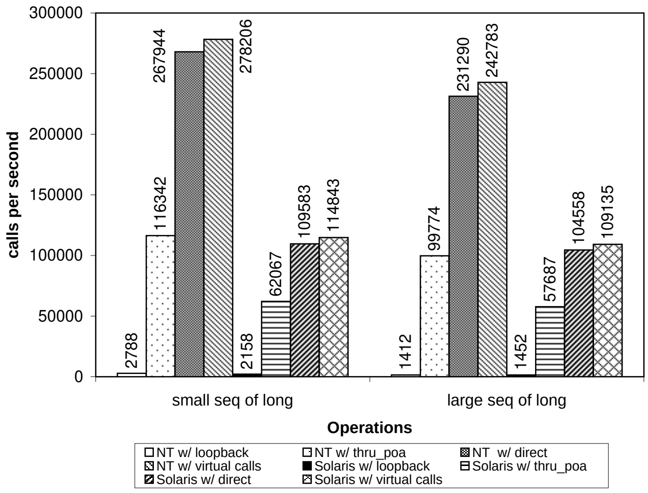

Figure 4: Results of TAO's Collocation Optimizations

As shown in Figure 4, the larger the size of arguments passed by the operations, the bigger the performance gain achieved by using collocation. Comparing the benchmarking results, note that there is 130%$\sim$180% performance gain when switching from the standard strategy to the direct strategy. This performance boost is achieved by skipping the various ORB and POA operations discussed earlier. The size of the arguments does not have a significant effect on the performance. Finally, if the operations are made directly by calling the skeleton methods, less than 5% of performance gain is incurred by avoiding an extra virtual method call.

These results illustrate that by using direct collocation optimizations, invocations on collocated CORBA objects are almost comparable to invocations on virtual methods of ordinary C++ objects.

6 Concluding Remarks

One of CORBA's early influences was Spring [9], which was designed from the ground up as an OO OS by Sun in the late '80s and early '90s. CORBA 1.x [10] borrowed heavily from Spring concepts and features, including its IDL syntax and semantics, its use of object references [11], and its strict separation of interface from implementation. Ironically, it took several years for CORBA implementations to adopt another key Spring feature, *collocation*, which can be used to transparently decouple the overhead of communicating with an object from how and when an object's servant(s) are configured into a server process.

In an earlier column [3], we illustrated how advanced POA features like Servant Managers can be used to configure and instantiate servants into server processes very late in the design lifecycle, *i.e.*, dynamically at run-time. The use of collocation makes these advanced POA features even more powerful since they allow the ORB to determine an optimal configuration transparently to applications.

The standard collocation strategy described in this column is completely CORBA compliant. It respects the availability of targeting objects and the threading policy that manages the targeting object. The direct collocation policy optimization is not entirely compliant with the CORBA standard, though it provides more efficient collocated operation invocations. However, both collocation strategies are much more efficient than using remote stubs that transmit data via the loopback network interface.

While collocation optimizations apply to normal object implementations, there is another class of object implementations that requires direct invocation only. These *locality-constrained* objects, such as `Policy` objects and interceptors, are often part of the ORB implementation itself. The Hewlett-Packard/IONA submission[12] to the OMG Portable Interceptor RFP[13], for example, defines `CORBA::LocalObject` as a `native` IDL type to be used as a language-mapping-specific implementation base class for servants for locality-constrained objects. Application-defined interceptors use `LocalObject` as a servant base class that provides local implementations for all `CORBA::Object` operations. Object references for these objects are simply direct C++ pointers to the servants. Such objects are referenced directly because they typically exist at levels below the POA, *e.g.*, within the request invocation path, and thus can't be implemented as POA-based servants.

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] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

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

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

[4] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[5] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[6] Object Management Group, *OMG Security Service*, OMG Document ptc/98-01-02, revision 1.2 ed., January 1998.

[7] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

[8] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Dynamic Scheduling Strategies for Avionics Mission Computing," in *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Nov. 1998.

[9] S. Radia, G. Hamilton, P. Kessler, and M. Powell, "The Spring Object Model," in *Proceedings of the Conference on Object-Oriented Technologies*, (Monterey, CA), USENIX, June 1995.

[10] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[11] M. L. Powell, *Objects, References, Identifiers, and Equality White Paper*. SunSoft, Inc., OMG Document 93-07-05 ed., July 1993.

[12] H.-P. Company and I. T. PLC, *Portable Interceptor RFP Initial Submission*. Object Management Group, OMG Document orbos/99-04-08 ed., April 1999.

[13] Object Management Group, *Portable Interceptor RFP*, OMG Document orbos/98-09-11 ed., September 1998.