An Overview of the CORBA Portable Object Adapter

Irfan Pyarali and Douglas C. Schmidt
{irfan,schmidt}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

This paper will appear in a special issue of the ACM StandardView magazine on CORBA.

Abstract

An Object Adapter is an integral part of the Common Object Request Broker Architecture (CORBA). An Object Adapter assists an Object Request Broker (ORB) in delivering client requests to server object implementations (servants). Services provided by an Object Adapter include: (1) generating and interpreting object references, (2) activating and deactivating servants, (3) demultiplexing requests to map object references onto their corresponding servants, and (4) collaborating with automatically-generated IDL skeletons to invoke operations on servants.

This paper provides two contributions to the study of Object Adapters. First, it outlines the CORBA Portable Object Adapter (POA) specification, which is a recent addition to the CORBA standard that greatly simplifies the development of portable and extensible servants and server applications. The design goals, architectural components, and semantics of the POA are explained. Second, the paper describes the design choices made to adapt the POA for the TAO Real-time ORB. Key design issues regarding efficient demultiplexing, upcall and collocation optimizations, ORB and POA concurrency configurations, POA synchronization, and predictability are covered.

1 Introduction

The Common Object Request Broker Architecture (CORBA) [1] is an emerging standard for distributed object computing (DOC) middleware. DOC middleware resides between clients and servers, simplifying application development by providing a uniform view of heterogeneous network and OS layers.

At the heart of DOC middleware are *Object Request Brokers* (ORBs), such as CORBA [1], DCOM [2], and Java RMI [3]. ORBs eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications by automating common network programming tasks such as object location, object activation, parameter marshaling, fault recovery, and security. Thus, ORBs facilitate the development of flexible distributed applications and reusable services in heterogeneous distributed environments.

The Portable Object Adapter (POA) specification [4] is an important new component that the OMG has defined for the CORBA standard. The POA is an integral part of the server-side of the CORBA reference model. It allows developers to construct CORBA server applications that are portable between heterogeneous ORB implementations [5].

This paper is organized as follows: Section 2 gives an overview of the CORBA architecture and shows how the Object Adapter fits into this architecture; Section 3 describes the functionality provided by a CORBA Object Adapter and introduces the POA [4]; Section 4 outlines the designed goals of the POA as specified by the OMG; Section 5 presents an overview of the POA architecture; Section 6 illustrates the key interactions and collaborations of POA components; Section 7 discusses the POA features necessary for a Real-time ORB; and Section 8 presents concluding remarks.

2 CORBA Architecture

CORBA Object Request Brokers (ORBs) [6] allow clients to invoke operations on distributed objects without concern for:

Object location: CORBA objects can be collocated with the client or distributed on a remote server, without affecting their implementation or use.

Programming language: The languages supported by CORBA include C, C++, Java, Ada95, COBOL, and Smalltalk, among others.

OS platform: CORBA runs on many OS platforms, including Win32, UNIX, MVS, and real-time embedded systems like VxWorks, Chorus, and LynxOS.

*This work was supported in part by Boeing, CDI, DARPA contract 9701516, Lucent, Motorola, NSF grant NCR-9628218, Siemens, and US Sprint.

Communication protocols and interconnects: The communication protocols and interconnects that CORBA can run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, embedded system backplanes, and shared memory.

Hardware: CORBA shields applications from side-effects stemming from differences in hardware such as storage layout and data type sizes/ranges.

Figure 1 illustrates the components in the CORBA reference model, all of which collaborate to provide the portability, interoperability, and transparency outlined above. Each component in the CORBA reference model is outlined below:

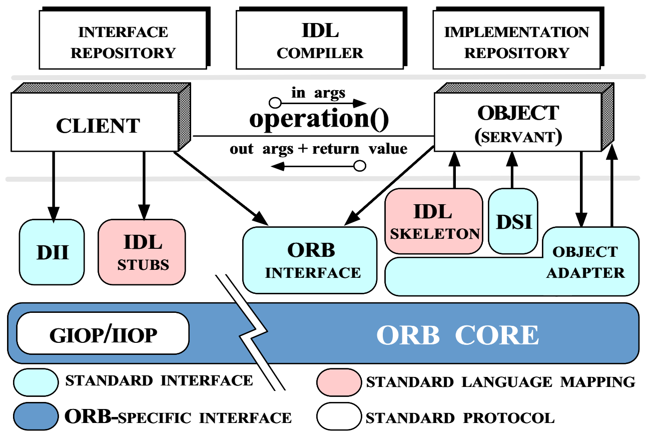

Figure 1: Components in the CORBA Reference Model

Client: This program entity performs application tasks by obtaining object references to objects and invoking operations on them. Objects can be remote or collocated relative to the client. Ideally, accessing a remote object should be as simple as calling an operation on a local object, *i.e.*, `object→operation(args)`. Figure 1 shows the underlying components that ORBs use to transmit remote operation requests transparently from client to object.

Object: In CORBA, an object is an instance of an Interface Definition Language (IDL) interface. The object is identified by an *object reference*, which uniquely names that instance across servers. An *ObjectId* associates an object with its servant implementation, and is unique within the scope of an Object Adapter. An object has one or more servants associated with it that implement the interface.

Servant: This component implements the operations defined by an OMG Interface Definition Language (IDL) interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are implemented using one or more objects. In non-OO languages like C, servants are typically implemented using functions and `structs`. A client never interacts with a servant directly, but always through an object.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. For objects executing remotely, a CORBA-compliant [1] ORB Core communicates via some version of the General Inter-ORB Protocol (GIOP), most commonly the Internet Inter-ORB Protocol (IIOP), which runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked into both client and server applications.

ORB Interface: An ORB is an abstraction that can be implemented various ways, *e.g.*, one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an interface to an ORB. This ORB interface provides standard operations that (1) initialize and shutdown the ORB, (2) convert object references to strings and back, and (3) create argument lists for requests made through the *dynamic invocation interface* (DII).

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a “glue” between the client and servants, respectively, and the ORB. Stubs provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common data-level representation. Conversely, skeletons demarshal the data-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler automatically transforms OMG IDL definitions into an application programming language like C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [7].

Dynamic Invocation Interface (DII): The DII allows clients to generate requests at run-time. This flexibility is useful when an application has no compile-time knowledge of the interface it is accessing. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of twoway operations to avoid blocking the client until the servant responds. In contrast, SII stubs currently only support *twoway*, *i.e.*, request/response, and *oneway*, *i.e.*, request only operations, though the OMG has standardized an asynchronous method invocation interface in the recent Messaging Service specification [8].

Dynamic Skeleton Interface (DSI): The DSI is the server’s analogue to the client’s DII. The DSI allows an ORB to deliver requests to a servant that has no compile-time knowledge of the IDL interface it is implementing. Clients making requests need not know whether the server ORB uses static skeletons or dynamic skeletons. Likewise, servers need not know if clients use the DII or SII to invoke requests.

Object Adapter: An Object Adapter associates a servant with objects, demultiplexes incoming requests to the servant, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on that servant. Recent CORBA portability enhancements [1] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can still support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

Interface Repository: The Interface Repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make invocations on it. In addition, the Interface Repository provides a common location to store additional information associated with interfaces ORB objects, such as stub/skeleton type libraries.

Implementation Repository: The Implementation Repository [9] contains information that allows an ORB to activate servers to process servants. Most of the information in the Implementation Repository is specific to an ORB or OS environment. In addition, the Implementation Repository provides a common location to store information associated with servers, such as administrative control, resource allocation, security, and activation modes.

3 Object Adapter Overview

This section describes the functionality provided by a CORBA Object Adapter. In addition, this section introduces the Portable Object Adapter (POA) and contrasts the POA with its predecessor, the Basic Object Adapter (BOA).

3.1 Object Adapter Functionality

A CORBA Object Adapter is responsible for: (a) generating object references, (b) activation and deactivation of servants, (c) demultiplexing requests to servants, and (d) collaborating with IDL skeletons to invoke servant operations. These responsibilities are described in detail below:

Generating object references: An Object Adapter is responsible for generating object references for the CORBA objects registered with it. Object references identify a CORBA object and contain addressing information that allow clients to to invoke operations on that object in a distributed system. Object Adapters cooperate with the communication mechanisms in the ORB Core and underlying OS to ensure that the information necessary to reach an object is present in the object reference.

Figure 2 shows a typical Interoperable Object Reference (IOR), which supports the Internet Inter-ORB Protocol (IIOP) [1]. An IOR contains the IIOP version, host name, and port number that identifies a communication endpoint for the server process; some means to ensure uniqueness for certain types of IORs, *e.g.*, timestamps for *transient* IORs; the identity of the Object Adapter; and the identity of the CORBA object.

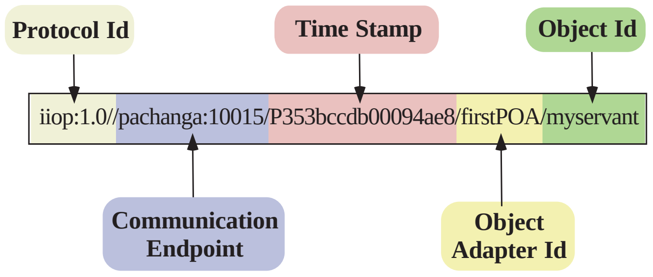

Figure 2: Interoperable Object Reference

Activation and deactivation of servants: Object Adapters can activate CORBA objects to handle client requests. To accomplish this, an Object Adapter can be programmed to create servants that handle requests for those objects. Similarly, Object Adapters can deactivate objects and can destroy their corresponding servants when they are no longer needed, *e.g.*, to reduce server memory consumption.

Demultiplexing requests to servants: Object Adapters demultiplex CORBA requests to the appropriate servants. When an ORB Core receives a request, it collaborates with the Object Adapter through a private, *i.e.*, non-standardized, interface to ensure that the request reaches the proper servant. The Object Adapter parses the request to locate the Object Id of the servant, which it uses to locate the correct servant and invoke the appropriate operation on the servant.

Invoking servant operations: The operation name is specified in the CORBA request. Once the Object Adapter locates the target servant, it dispatches the requested operation on the servant. Before the request is invoked on the servant, however, the Object Adapter uses an IDL skeleton to transform the parameters in the request into arguments. The skeleton then passes the demarshaled arguments as parameters to the intended servant operation.

3.2 Portable Object Adapter (POA)

The Portable Object Adapter (POA) is a standard component in the CORBA model recently specified by the OMG [4].

The POA allows programmers to construct servants that are portable between different ORB implementations. Portability is achieved by standardizing the skeletons classes produced by the IDL compiler, as well as the interactions between the servants and the Object Adapter.

The POA's predecessor was the Basic Object Adapter (BOA). The BOA was widely recognized to be incomplete and underspecified. For instance, the API for registering servants with the BOA was unspecified. Therefore, different implementors made many interpretations and extensions to provide a complete ORB. These interpretations and extensions were incompatible with each other, however, and there was no simple upgrade to the BOA that made existing applications portable.

The solution adopted by the OMG was to abandon the BOA and create a new Object Adapter that *was* portable. ORB implementors can maintain their proprietary BOA to support their current customer base. Existing programs continue to work and are supported by their ORB vendors. In the future, the OMG will no longer include the BOA with the CORBA specification.

4 The POA Design Goals

The OMG's design goals for the Portable Object Adapter (POA) specification include the following:

Portability: The POA allows programmers to construct servants that are portable between different ORB implementations. Hence, the programmer can switch ORBs without having to modify existing servant code. The lack of this feature was a major shortcoming of the Basic Object Adapter (BOA).

Persistent identities: The POA supports objects with persistent identities. More precisely, the POA is designed to support servants that can provide consistent service for objects whose lifetimes span multiple server process lifetimes.

Automation: The POA supports transparent activation of objects and implicit activation of servants. This automation makes the POA easier and simpler to use.

Conserving resources: There are many situations where a server must support many CORBA objects. For example, a database server that models each database record as a CORBA object can potentially service hundreds of objects. The POA allows a single servant to support multiple Object Ids simultaneously. This allows one servant to service many CORBA objects, thereby conserving memory resources on the server.

Flexibility: The POA allows servants to assume complete responsibility for an object's behavior. For instance, a servant can control an object's behavior by defining the object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing code that will be executed in response to requests, and determining whether or not the object exists at any point in time.

Behavior governed by policies: The POA provides an extensible mechanism for associating policies with servants in a POA. Currently, the POA supports seven policies, such as threading, retention, and lifespan policies, that can be selected at POA creation time. An overview of these policies is presented in Section 5.

Nested POAs: The POA allows multiple distinct, nested instances of the POA to exist in a server. Each POA in the server provides a namespace for all the objects registered with that POA and all the child POAs that are created by this POA. The POA supports recursive deletes, *i.e.*, destroying a POA destroys all its child POAs.

SSI and DSI support: The POA allows programmers to construct servants that inherit from (1) static skeleton classes (SSI) generated by OMG IDL compilers or (2) a Dynamic Skeleton Interface (DSI). Clients need not be aware that a CORBA object is serviced by a DSI servant or an IDL servant. Two CORBA objects supporting the same interface can be serviced one by a DSI servant and the other with an IDL servant. Furthermore, a CORBA object may be serviced by a DSI servant during some period of time, while the rest of the time is serviced by an IDL servant.

5 The POA Architecture

The ORB is an abstraction visible to both the client and server. In contrast, the POA is an ORB component visible only to the server, *i.e.*, clients are not directly aware of the POA's existence or structure. This section describes the architecture of the request dispatching model defined by the POA and the interactions between its standard components and the ORB Core.

User-supplied servants are registered with the POA.[1] Clients hold object references upon which they make requests, which the POA ultimately dispatches as operations on a servant. The ORB, POA, servant, and skeleton all collaborate to determine (1) which servant the operation should be invoked on and (2) to dispatch the invocation.

Figure 3 shows the POA architecture. As shown in this figure, a distinguished POA, called the `Root POA`, is created and managed by the ORB. The `Root POA` is always available to an application through the ORB initialization interface, `resolve_initial_references`. The application developer can register servants with the `Root POA` if the policies

[1]This statement is a simplification – more detail is provided below.

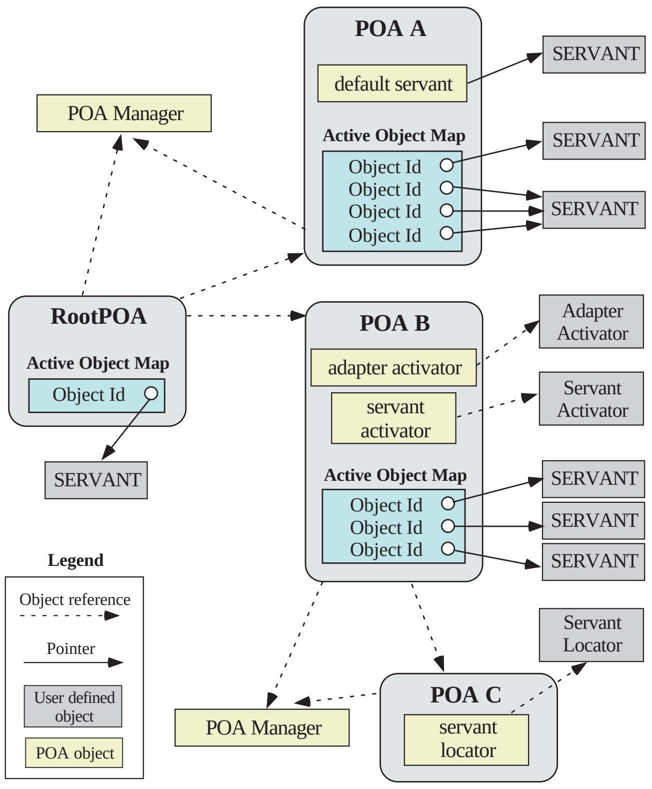

Figure 3: POA Architecture

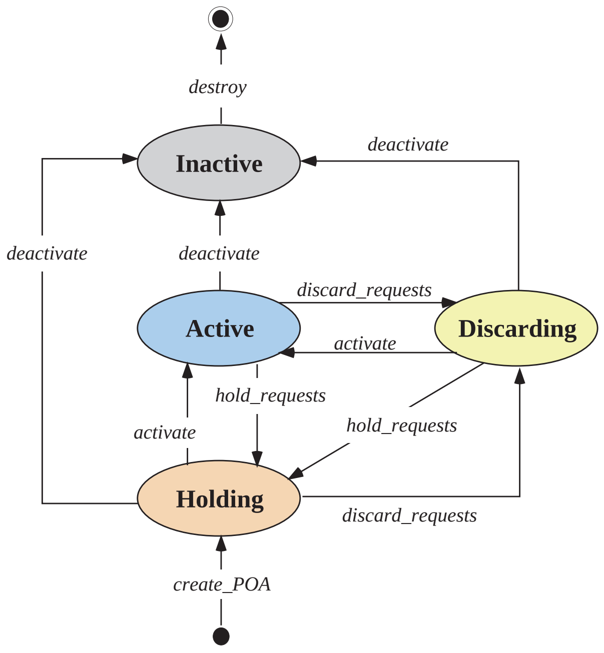

Figure 4: POA Manager Processing States

of the `Root POA` specified in the POA specification are suitable for the application.

A server application may want to create multiple POAs to support different kinds of CORBA objects and/or different kinds of servant styles. For example, a server application might have two POAs: one supporting transient CORBA objects and the other supporting persistent CORBA objects [10]. A nested POA can be created by invoking the `create_POA` factory operation on a parent POA.

The server application in Figure 3 contains three other nested POAs: `A`, `B`, and `C`. POA `A` and `B` are children of the `Root POA`; POA `C` is `B`'s child. Each POA has an *Active Object Table* that maps Object Ids to servants. Other key components in a POA are described below:

POA Manager: A POA manager encapsulates the processing state of one or more POAs. By invoking operations on a POA manager, server applications can cause requests for the associated POAs to be queued or discarded. In addition, applications can use the POA manager to deactivate POAs. Figure 4 shows the processing states of a POA Manager and the operations required to transition from one state to another.

Adapter Activator: An adapter activator can be associated with a POA by an application. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not yet exist. The adapter activator can then decide whether or not to create the required POA on demand.

For example, if the target object reference was created by a POA whose full name is `/A/B/C` and only POA `/A` and POA `/A/B` currently exist, the `unknown_adapter` operation will be invoked on the adapter activator associated with POA `/A/B`. In this case, POA `/A/B` will be passed as the parent parameter and `C` as the name of the missing POA to the `unknown_adapter` operation.

Servant Manager: A servant manager is a locality constrainted servant that server applications can associate with a POA [11]. The ORB uses a servant manager to activate servants on demand, as well as to deactivate servants. Servant managers are responsible for (1) managing the association of an object (as characterized by its Object Id value) with a particular servant and (2) for determining whether an object exists or not. There are two types of servant managers: `ServantActivator` and `ServantLocator`. The type used in a particular situation depends on the policies in a POA, which are described next.

POA Policies: The characteristics of each POA other than the `Root POA` can be customized at POA creation time using different *policies*. The policies of the `Root POA` are specified in the POA specification. The POA specification defines the following policies:

- **Threading policy:** This policy is used to specify the threading model used with the POA. A POA can either be single-threaded or have the ORB control its threads. If it is single-threaded, all requests are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code, *i.e.*, servants and servant managers, are invoked in a manner that is safe for code that is unaware of multi-threading.

  In contrast, if the ORB-controlled threading policy is specified, the ORB determines the thread (or threads) that the POA dispatches its requests in. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.

- **Lifespan policy:** This policy is used to specify whether the CORBA objects created within a POA are persistent or transient. Persistent objects can outlive the process in which they are created initially. In contrast, transient objects cannot outlive the process in which they are created initially. Once the POA is deactivated, use of any object references generated for a transient object will result in an `CORBA::OBJECT_NOT_EXIST` exception.

- **Object Id uniqueness policy:** This policy is used to specify whether the servants activated in the POA must have unique Object Ids. With the unique Id policy, servants activated with that POA support exactly one Object Id. However, with the multiple Id policy, a servant activated with that POA may support one or more Object Ids.

- **ObjectId assignment policy:** This policy is used to specify whether Object Ids in the POA are generated by the application or by the ORB. If the POA also has the persistent lifespan policy, ORB assigned Object Ids must be unique across all instantiations of the same POA.

- **Implicit activation policy:** This policy is used to specify whether implicit activation of servants is supported in the POA. A C++ server can create a servant, and then by setting its POA and invoking its `_this` method, it can register the servant implicitly and create an object reference in a single operation.

- **Servant retention policy:** This policy is used to specify whether the POA retains active servants in an *Active Object Map*. A POA either retains the associations between servants and CORBA objects or it establishes a new CORBA object/servant association for each incoming request.

- **Request processing policy:** This policy is used to specify how requests should be processed by the POA. When a request arrives for a given CORBA object, the POA can do one of the following:

  - *Consult its Active Object Map only* – If the Object Id is not found in the Active Object Map, the POA returns an `CORBA::OBJECT_NOT_EXIST` exception to the client.

  - *Use a default servant* – If the Object Id is not found in the Active Object Map, the request is dispatched to the default servant (if available).

  - *Invoke a servant manager* – If the Object Id is not found in the Active Object Map, the servant manager (if available) is given the opportunity to locate a servant or raise an exception. The servant manager is an application-supplied object that can incarnate or activate a servant and return it to the POA for continued request processing. Two forms of servant manager are supported: `ServantActivator`, which is used for a POA with the `RETAIN` policy, and `ServantLocator`, which is used with the `NON_RETAIN` policy.

Combining these policies with the retention policies described above provides the POA with a great deal of flexibility. Section 6 provides further details on how the POA processes requests.

6 The POA Semantics

The POA is used primarily in two modes: (1) request processing and (2) the activation and deactivation of servants and objects. This section describes these two modes and outlines the semantics and behavior of the interactions that occur between the components in the POA architecture.

6.1 Request Processing

Each client request contains an *Object Key*. The Object Key conveys the Object Id of the target object and the identity of the POA that created the target object reference. The end-to-end processing of a client request occurs in the follow steps:

1. Locate the server process: When a client issues a request, the ORB first locates an appropriate server process, using the Implementation Repository to create a new process if necessary. In an ORB that uses IIOP, the host name and port number in the Interoperable Object Reference (IOR) identify the communication endpoint of the server process.

2. Locate the POA: Once the server process has been located, the ORB locates the appropriate POA within that server. If the designated POA does not exist in the server process, the server has the opportunity to re-create the required POA by using an adapter activator. The name of the target POA is specified by the IOR in a manner that is opaque to the client.

3. Locate the servant: Once the ORB has located the appropriate POA, it delivers the request to that POA. The POA finds the appropriate servant by following its servant retention and request processing policies, which are described in Section 5.

4. Locate the skeleton: The final step the POA performs is to locate the IDL skeleton that will transform the parameters in the request into arguments. The skeleton then passes the demarshaled arguments as parameters to the correct servant operation, which it locates via one of the operation demultiplexing strategies described in Section 7.1.

5. Handling replies, exceptions and location forwarding: The skeleton marshals any exceptions, return values, `inout`, and `out` parameters returned by the servant so that they can be sent to the client. The only exception that is given special treatment is the `ForwardRequest` exception. It causes the ORB to deliver the current request and subsequent requests to the object denoted in the `forward_reference` member of the exception.

6.2 Object Reference Creation

Object references are created in servers. Object references encapsulate Object Id and other information required by the ORB to locate the server and POA with which the object is associated, *e.g.*, in which POA scope the reference was created. Object references can be created in the following ways:

Explicit creation of object references: A server application can directly create a reference with the `create_reference` and `create_reference_with_id` operations on a POA object. These operations only create a reference, but do not associate the designated object with an active servant.

Explicit activation of servants: A server application can activate a servant explicitly by associating it with an Object Id using the `activate_object` or `activate_object_with_id` operations. Once activated, the server application can map the servant to its corresponding reference using the `servant_to_reference` or `id_to_reference` operations.

Implicit activation of servants: If the server application attempts to obtain an object reference corresponding to an inactive servant and the POA supports the implicit activation policy, the POA can automatically assign a generated unique Object Id to the servant and activate the resulting object.

Once a reference is created in the server, it can be exported to clients in a variety of ways. For instance, it can be advertised via the OMG Naming and Trading Services. Likewise, it can be converted to a string via `CORBA::object_to_string` and published in some way that allows the client to discover the string and convert it to a reference using `CORBA::string_to_object`. Moreover, it can be returned as the result of an operation invocation, *i.e.*, using the *factory method* pattern [12]. Regardless of how an object reference is obtained, however, once a client has an object reference it can invoke operations on the object.

7 Designing a POA for Real-time ORBs

The Distributed Object Computing group at Washington University has developed a high-performance, real-time ORB endsystem called The ACE ORB (TAO) [13]. TAO provides end-to-end quality of service guarantees to applications by vertically integrating CORBA middleware with OS I/O subsystems, communication protocols, and network interfaces.

To adapt the CORBA Portable Object Adapter (POA) specification into TAO, certain architectural considerations were necessary to fulfill real-time requirements. This section outlines these considerations, describes the design patterns we applied to maximize the predictability and performance of TAO's POA, and provides references to information on TAO's design and performance results.

7.1 Efficient Request Demultiplexing

7.1.1 Conventional ORB Demultiplexing Strategies

A standard GIOP-compliant client request contains the identity of its remote object and remote operation. A remote object is represented by an Object Key `octet sequence` and a remote operation is represented as a `string`. Conventional ORBs demultiplex client requests to the appropriate operation of the servant implementation using the *layered demultiplexing* architecture shown in Figure 5. These steps perform the following tasks:

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.*, through the data link, network, and transport layers up to the user/kernel boundary and the ORB Core.

Steps 3, 4, and 5: The ORB Core uses the addressing information in the client's Object Key to locate the appropriate Object Adapter, servant, and the skeleton of the target IDL operation.

Step 6: The IDL skeleton locates the appropriate operation, demarshals the request buffer into operation parameters, and performs the operation upcall.

However, layered demultiplexing is generally inappropriate for high-performance and real-time applications for the following reasons [14]:

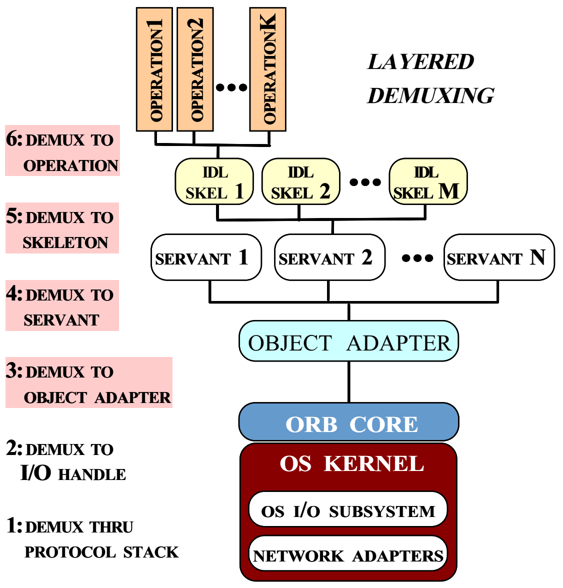

Figure 5: Layered CORBA Request Demultiplexing

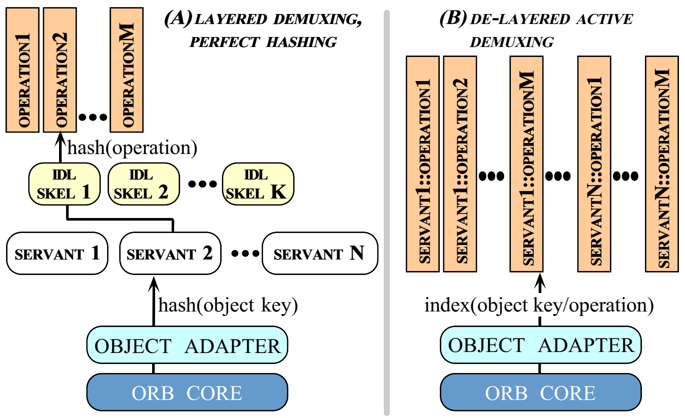

Figure 6: Optimized CORBA Request Demultiplexing Strategies

Decreased efficiency: Layered demultiplexing reduces performance by increasing the number of internal tables that must be searched as incoming client requests ascend through the processing layers in an ORB endsystem. Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an Object Adapter.

Increased priority inversion and non-determinism: Layered demultiplexing can cause priority inversions because servant-level quality of service (QoS) information is inaccessible to the lowest-level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem. Therefore, an Object Adapter may demultiplex packets according to their FIFO order of arrival. FIFO demultiplexing can cause higher priority packets to wait for an indeterminate period of time while lower priority packets are demultiplexed and dispatched [15].

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [16, 17] show that conventional ORBs spend $\sim$17% of the total server time processing demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide uniform, scalable QoS guarantees to real-time applications.

7.1.2 TAO's Optimized ORB Demultiplexing Strategies

To address the limitations with conventional ORBs, TAO provides the demultiplexing strategies shown in Figure 6. TAO's optimized demultiplexing strategies include the following:

Perfect hashing: The perfect hashing strategy shown in Figure 6(A) is a two-step layered demultiplexing strategy. This strategy uses an automatically-generated perfect hashing function to locate the servant. A second perfect hashing function is then used to locate the operation. The primary benefit of this strategy is that servant and operation lookups require $O(1)$ time in the worst-case.

TAO uses the GNU `gperf` [18] tool to generate perfect hash functions for object keys and operation names. This perfect hashing scheme is applicable when the keys to be hashed are known *a priori*. In many deterministic real-time systems, such as avionic mission control systems [19], the servants and operations can be configured statically. For these applications, it is possible to use perfect hashing to locate servants and operations.

Active demultiplexing: TAO also provides a more dynamic demultiplexing strategy called *active demultiplexing*, shown in Figure 6(B). In this strategy, the client passes a handle that directly identifies the servant and operation in $O(1)$ time in the worst-case. This handle can be configured into a client when it obtains a servant's object reference, *e.g.*, via a Naming service or Trading service. Once the request arrives at the server ORB, the Object Adapter uses the handle supplied in the CORBA request header to locate the servant and its associated operation in a single step.

Unlike perfect hashing, TAO's active demultiplexing strategy does not require that all Object Ids be known *a priori*. This makes it more suitable for applications that incarnate and etherealize CORBA objects dynamically.

Both perfect hashing and active demultiplexing can demultiplex client requests efficiently and predictably. Moreover, these strategies perform optimally regardless of the number of active connections, application-level servant implementations, and operations defined in IDL interfaces. [20] presents a detailed study of these and other request demultiplexing strategies for a range of target objects and operations.

TAO's Object Adapter uses the Service Configurator pattern [21] to select perfect hashing or active demultiplexing dynamically at ORB installation-time [22]. Both of these strategies improve request demultiplexing performance and predictability above the ORB Core.

To facilitate various strategies for finding and dispatching servants, TAO uses the active object map class hierarchy shown in Figure 7. This design is an example of the Bridge

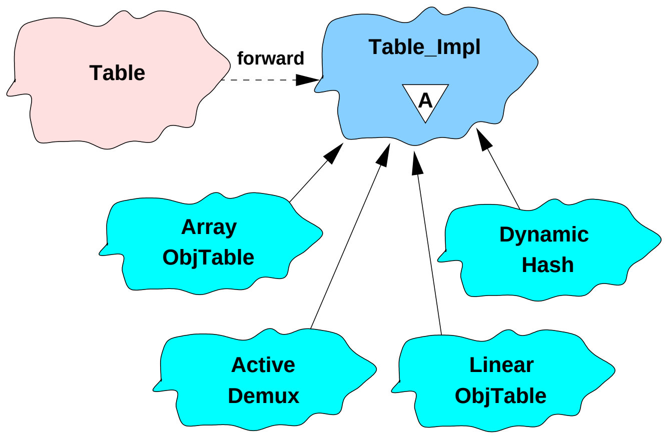

Figure 7: Class Hierarchy of POA Active Object Maps

and Strategy patterns [12], where the interface of the map is decoupled from its implementation so that the two can vary independently.

7.2 Supporting Custom ORB Core and POA Configurations

An increasingly important class of distributed applications require stringent quality of service (QoS) guarantees. These applications include *telecommunication systems* such as call processing and switching; *command and control systems* such as avionics mission control programs and tactical shipboard computing; *multimedia* such as video-on-demand and teleconferencing; and *simulations* such as battle readiness planning.

In order for ORB middleware to support real-time application QoS requirements, it must be adaptable and configurable. To achieve this, TAO supports various server configurations, including different ORB Core configurations that allow applications to customize request processing and the management of transport connections. For instance, TAO's ORB Core can be configured to process all requests in one thread, each request in a separate thread, or each connection in a separate thread.

To ensure consistent behavior throughout the layers in an ORB endsystem, TAO's POA is designed to support TAO's various ORB Core configurations. The important variations are (1) each ORB Core in a process has its own POA and (2) all ORB Cores in a process share one POA, as described below:

POA per ORB Core: Figure 8 shows this ORB configuration, where each ORB Core in a server process maintains a

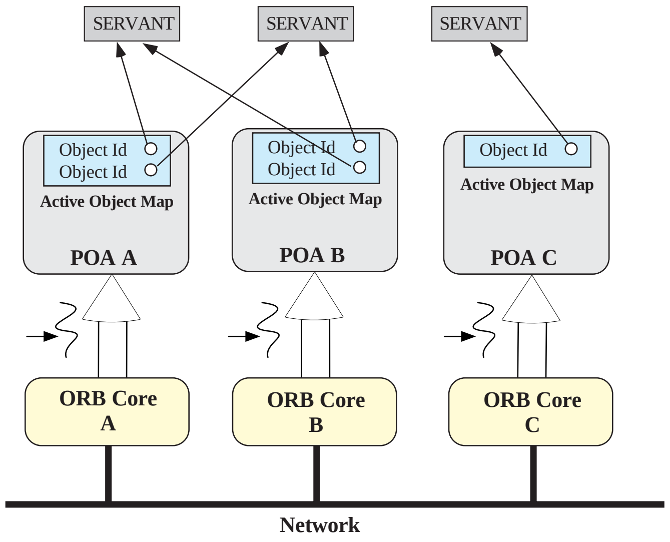

Figure 8: POA-per-ORB Configuration

distinct POA instance. This configuration is generally chosen for deterministic real-time applications, such as avionics mission computing [19], where each ORB Core has its own thread of control that runs at a distinct priority.

When this configuration is used, each POA is not accessed by other threads in the process. Thus, no locking is required within a POA, thereby reducing the overhead and nondeterminism incurred to dispatch servant requests. The drawback of this configuration, however, is that registering servants becomes more complicated if servants must be registered in multiple POAs.

Global POA: Figure 9 show this ORB configuration, where all ORB Cores in a server process share the same POA instance. The main benefit of this configuration is that servant registration is straightforward since there is only one POA. However, the drawback is that this POA requires additional locks since it is shared by all the threads in the process. These threads may simultaneously change the state of active object maps in the POA by adding and removing servants.

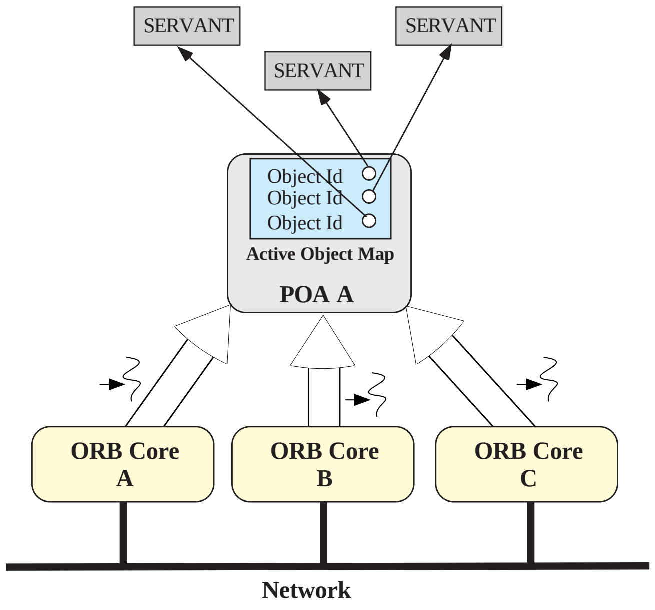

Figure 9: Global POA Configuration

7.3 POA Synchronization

TAO has been designed to minimize synchronization in the critical request processing path of the ORB in order to improve its predictability and maximize its performance. Under certain ORB configurations, no synchronization is required in a POA. For example, if only one thread uses a POA, as described in Section 7.2, there is no need for mutual exclusion in the POA. Likewise, no synchronization is needed if the state of a POA does not changed during the execution of a server. This situation can happen when all the servants and servant managers are registered at server startup and no dynamic registrations occur at run-time.

To enable applications to select the most efficient POA synchronization, TAO's POA contains the following POA creation policy extensions:

```
// IDL
enum SynchronizationPolicyValue
{
  NULL_LOCK, THREAD_LOCK, DEFAULT_LOCK
};

interface SynchronizationPolicy
  : CORBA::Policy
{
  readonly attribute
    SynchronizationPolicyValue value;
};

SynchronizationPolicy create_synchronization_policy
  (in SynchronizationPolicyValue value);
```

Objects that support the `SynchronizationPolicy` interface can be obtained using the POA's `create_synchronization_policy` operation. They are passed to the `POA::create_POA` operation to specify the synchronization used in the created POA. The `value` attribute of `SynchronizationPolicy` contains the value supplied to the `create_synchronization_policy` operation from which it was obtained. The following values can be supplied:

NULL_LOCK: No synchronization will be used to protect the internal state of the POA. This option should be used when the state of the created POA will not change during the execution of the server or when only one thread will use the POA.

THREAD_LOCK: The internal state of the POA will be protected against simultaneous changes from multiple threads. This option should be used when multiple threads will use the POA simultaneously.

DEFAULT_LOCK: The ORB configuration file will be consulted to determine whether to use a thread lock or null lock. This option should be used when the server programmer wants to delay the POA synchronization choice until run-time.

If no `SynchronizationPolicy` object is passed to `create_POA`, the synchronization policy defaults to `DEFAULT_LOCK`. The `DEFAULT_LOCK` option allows applications to make the synchronization decision once for all the POAs created in the server. For example, if the server is single threaded, the application can specify in the configuration file that the default lock should be the null lock. Hence, the application does not have to specify the `NULL_LOCK` policy in every call to `create_POA`.

Figure 10 shows the class hierarchy of the POA locks. The locking strategies used in TAO's POA are an example of the External Polymorphism pattern [23], where C++ classes unrelated by inheritance and/or having no virtual methods can be treated polymorphically.

7.4 POA Dispatching Optimizations

The POA is in the critical request processing path of a server ORB. Thus, TAO performs the following upcall and collocation optimizations to reduce run-time processing overhead and jitter:

Upcall optimizations: Figure 11 shows a naive way to parse an object key. The Object Key is parsed and the individual fields of the key are stored in their respective objects. The problem with this approach is that it requires memory allocation for the individual objects and data copying to move the Object Key fields to the individual objects. Both of these operations increase POA overhead.

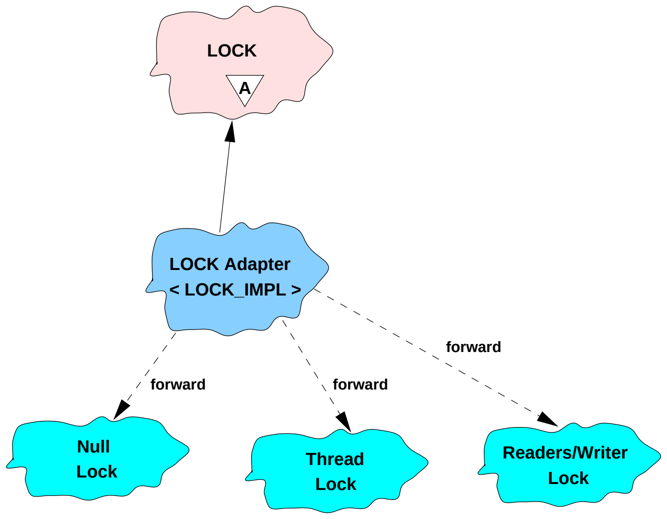

Figure 10: Class Hierarchy of POA Locks

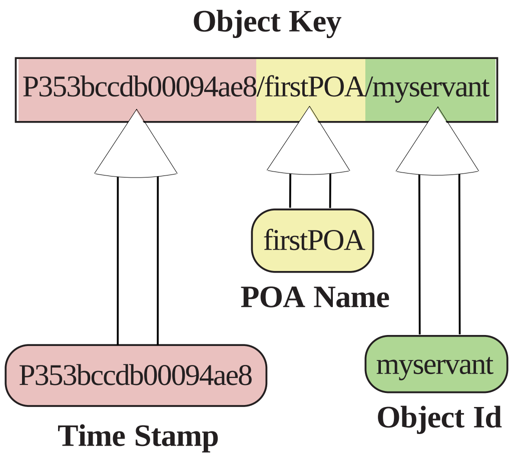

Figure 11: Naive Parsing of Object Key

Certain optimizations are possible during request dispatching in the POA. TAO takes advantage of the fact that the Object Key is available through the entire upcall. Thus, it does not modify the contents of the Object Key and the objects for the individual portions of the Object Key can be optimized to point to the correct locations in the Object Key. This approach is shown in Figure 12.

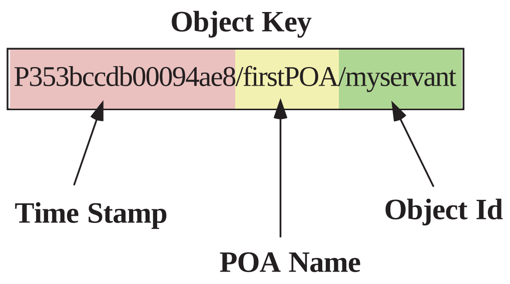

Figure 12: Optimized Parsing of Object Key

Collocation optimizations: One of the key strengths of CORBA is that it decouples (1) the implementation of servants from (2) how servants are configured into server processes throughout a distributed system. CORBA is largely used for communication between remote objects. However, there are configurations where a client and servant must be collocated in the same address space [24]. In this case, there is no need to marshal data or transmit operations through a "loopback" device.

TAO's POA optimizes for collocated client/servant configurations by generating a special stub for the client. This stub forwards all requests to the servant. Figure 13 shows the classes produced by the TAO IDL compiler.

The stub and skeleton classes shown in Figure 13 are required by the POA specification; the collocation class is specific to TAO. This feature is entirely transparent since the client only uses the abstract interface and never uses the collocation class directly. Therefore, the POA provides the collocation class, rather than the regular stub class when the servant is in the same address space of the client.

7.5 Predictability

Guaranteeing end-to-end predictability in TAO requires the POA to avoid calling external, unpredictable operations such as calling a servant locator for an incoming request. Hence, for TAO the following features of the POA can be disabled:[2]

[2]The emerging Real-time CORBA standard [25] also specifies that these POA features can be disabled for real-time applications.

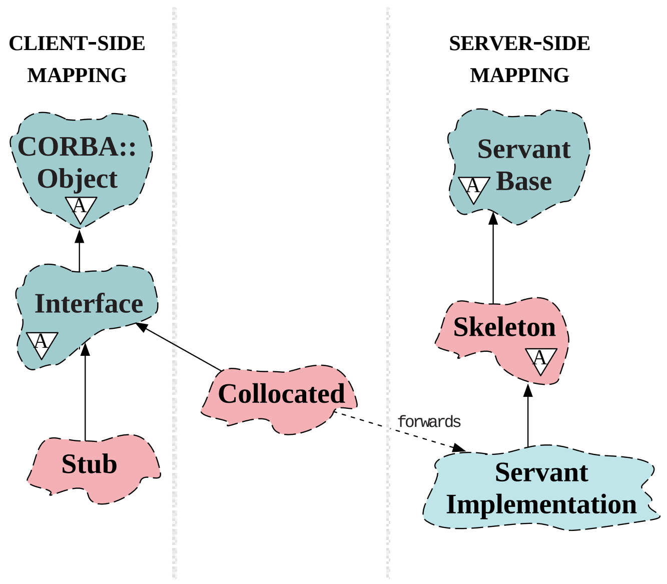

Figure 13: POA Mapping and Collocation Class

Servant Managers are not required: There is no need to locate servants in a real-time environment since all servants must be registered with POAs *a priori*.

Adapter Activators are not required: Real-time applications create all their POAs at the beginning of execution. Therefore, they need not use or provide an adapter activator. The alternative is to create POAs during request processing, in which case guarantees of end-to-end predictability are harder to achieve.

POA Managers are not required: The POA must not introduce extra levels of queuing in the ORB. Queuing can lead to priority inversion and extra locking. Therefore, the POA Manager in TAO can be disabled.

Our previous experience [20, 16, 26, 27, 28] measuring the performance of CORBA implementations showed that TAO supports efficient and predictable QoS better than other ORBs.

8 Concluding Remarks

A CORBA Object Adapter provides the following functionality: it (1) generates and interprets object references, (2) activates and deactivates servants, (3) demultiplexes requests to map object references onto their corresponding servants, and (4) collaborates with the automatically-generated IDL skeletons to invoke operations on servants.

OMG's new Portable Object Adapter (POA) specification defines a wide range of standard policies that enable developers to tailor an ORB's behavior to meet many different application use-cases. Although the POA is a relatively recent addition to the OMG CORBA model, it builds on the experience of the users and designers of the the Basic Object Adapter (BOA) [29] and other Object Adapters, such as OO Database Adapters [30]. In general, the POA is much more powerful and portable than the BOA, however.

In this paper, we provided a detailed discussion of the design and implementation of OMG's POA. In addition, we explained the features and optimizations necessary for a POA to work with the TAO real-time ORB. The implementation of the POA described in this paper is available at `www.cs.wustl.edu/~schmidt/TAO.html`. TAO has the first implementation of the POA specification in the CORBA industry.

Acknowledgements

Thanks to the DOC group at Washington University, especially Carlos O'Ryan, Aniruddha Gokhale, James Hu, and Sumedh Mungee for helping with the implementation of the POA and TAO, as well as providing comments on this paper.

References

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

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

[3] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[4] Object Management Group, *Specification of the Portable Object Adapter (POA)*, OMG Document orbos/97-05-15 ed., June 1997.

[5] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology," *C++ Report*, vol. 11, November/December 1997.

[6] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[7] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

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

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

[10] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *C++ Report*, vol. 12, April 1998.

[11] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter," *C++ Report*, vol. 12, Sept. 1998.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[13] 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.

[14] D. L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[15] D. C. Schmidt, F. Kuhns, R. Bector, and D. L. Levine, "The Design and Performance of an I/O Subsystem for Real-time ORB Endsystem Middleware," *submitted to the International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*.

[16] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[17] A. Gokhale and D. C. Schmidt, "Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks," in *Proceedings of the International Conference on Distributed Computing Systems*, (Baltimore, Maryland), IEEE, May 1997.

[18] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[19] 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.

[20] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[21] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[22] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *Submitted to the IEEE Communications Magazine*, 1998.

[23] C. Cleeland, D. C. Schmidt, and T. Harrison, "External Polymorphism – An Object Structural Pattern for Transparently Extending Concrete Data Types," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[24] D. C. Schmidt and S. Vinoski, "Developing C++ Servant Classes Using the Portable Object Adapter," *C++ Report*, vol. 12, June 1998.

[25] Object Management Group, *Realtime CORBA 1.0 Initial RFP Submission*, OMG Document orbos/98-01-08 ed., January 1998.

[26] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, (London, England), pp. 50–56, IEEE, November 1996.

[27] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[28] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[29] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[30] IONA, "IONA's Object Database Framework Adapter (ODAF)." www-usa.iona.com/Press/PR/odaf.html, 1997.