Applying Patterns to Develop Extensible ORB Middleware

Douglas C. Schmidt and Chris Cleeland
{schmidt,cleeland}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, USA
(314) 935-7538*

This article has been submitted to IEEE Communications Magazine, special issue on design patterns.

Abstract

Distributed object computing forms the basis for next-generation application middleware. At the heart of distributed object computing are Object Request Brokers (ORBs), which automate many tedious and error-prone distributed programming tasks. This article presents a case study of key design patterns needed to develop ORBs that can be dynamically configured and evolved for specific application requirements and system characteristics.

1 Introduction

Four trends are shaping the future of commercial software development. First, the software industry is moving away from *programming* applications from scratch to *integrating* applications using reusable components [1] such as ActiveX and Java Beans. Second, there is great demand for *distribution technology* such as remote method invocation or message-oriented middleware that simplifies application collaboration within and across enterprises [2]. Third, there are increasing efforts to define standard software infrastructures such as CORBA that allow applications to interwork seemlessly throughout *heterogeneous* environments [3]. Finally, next-generation distributed applications such as video-on-demand, teleconferencing, and avionics require Quality of Service (QoS) guarantees for latency, bandwidth, and reliability [4].

A key software technology supporting these trends is *distributed object computing (DOC) middleware*. DOC middleware facilitates the collaboration of local and remote application components in heterogeneous distributed environments. The goal of DOC middleware is to eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications. In particular, DOC middleware automates common network programming tasks such as object location, object activation, parameter marshaling, fault recovery, and security. At the heart of DOC middleware are *Object Request Brokers* (ORBs), such as CORBA [5], DCOM [6], and Java RMI [7].

This article describes how *design patterns* can be used to develop dynamically configurable ORB middleware that can be extended and maintained more easily than statically configured middleware. A pattern represents a recurring solution to a software development problem within a particular context [8]. Patterns help to alleviate the continual re-discovery and re-invention of software concepts and components by documenting proven solutions to standard software development problems [9]. For instance, patterns are useful for documenting the structure and participants in common communication software micro-architectures like Reactors [10], Active Objects [11], and Brokers [12]. These patterns are generalizations of object-structures that have been used successfully to build flexible and efficient event-driven and concurrent communication software such as ORBs.

To focus the discussion, this article presents a case study that illustrates how we have applied patterns to develop *The ACE ORB* (TAO) [13]. TAO is a freely available, highly extensible ORB targeted for applications with real-time quality of service (QoS) requirements. These applications include avionics mission computers [14], telecommunication switch management systems [10], and electronic medical imaging systems [15]. A novel aspect of TAO is its extensible design, which can be customized dynamically to meet specific application QoS requirements and network/endsystem characteristics.

*This research is supported in part by grants from Boeing, NSF grant NCR-9628218, Siemens, and Sprint.

2 Overview of CORBA and TAO

2.1 Overview of the CORBA ORB Reference Model

CORBA Object Request Brokers (ORBs) [3] allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware. Figure 1 illustrates the components in the CORBA reference model, which collaborate to provide the portability, interoperability, and transparency mentioned above.

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

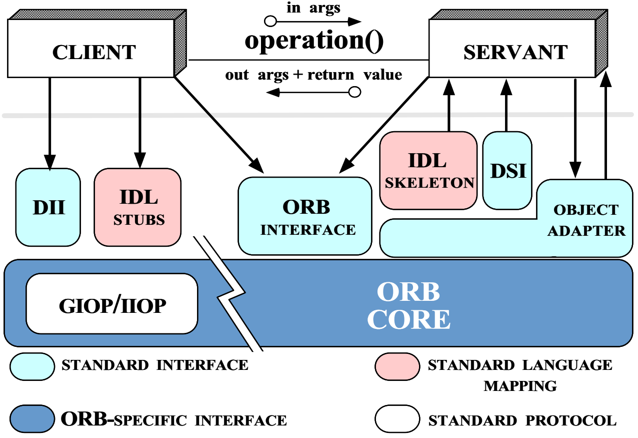

Figure 1: Components in the CORBA Reference Model

The client-side stubs marshal client operations into requests that are transmitted to servants via the standard Internet Inter-ORB Protocol (IIOP) implemented in the ORB Core. The server-side ORB Core receives these requests and passes them to the Object Adapter. The Object Adapter demultiplexes the requests to skeletons, which demarshal the requests and dispatch the appropriate application-level servant operation.

2.2 Overview of TAO

TAO is an ORB endsystem that contains the network interface, operating system, communication protocol, and CORBA middleware components and features shown in Figure 2. TAO is based on the standard OMG CORBA reference model, with the following enhancements designed to overcome the shortcomings of conventional ORBs for high-performance and real-time applications:

Real-time IDL Stubs and Skeletons: In addition to marshaling and demarshaling of operation parameters, TAO's

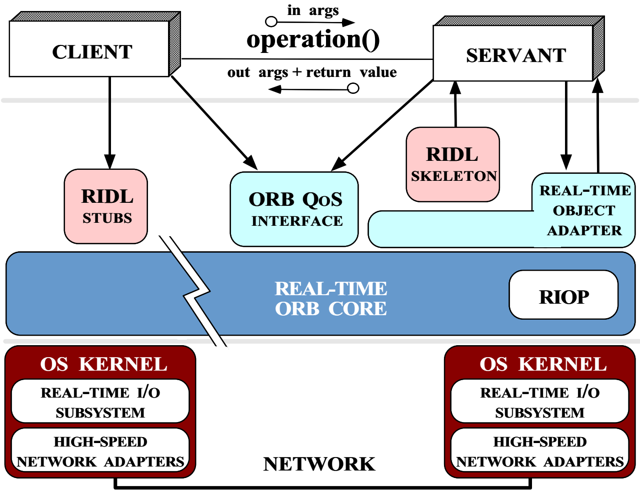

Figure 2: Components in the TAO Real-time ORB

Real-time IDL (RIDL) stubs and skeletons are designed to ensure that application timing requirements are specified and enforced end-to-end [16].

Real-time Object Adapter and ORB Core: In addition to associating servants with the ORB and demultiplexing incoming requests to servants, TAO's Object Adapter (OA) implementation dispatches servant operations in accordance with various real-time scheduling strategies such as Rate Monotonic and Maximal Urgency First [13].

ORB QoS Interface: TAO's QoS interface is designed to map real-time processing requirements to ORB endsystem/network resources. Common real-time processing requirements include end-to-end latency bounds and periodic scheduling deadlines. Common ORB endsystem/network resources include CPU, memory, network connections and storage devices.

Real-time I/O subsystem: TAO's real-time I/O subsystem performs admission control and assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be guaranteed.

High-speed network adapters: TAO's I/O subsystem contains a "daisy-chained" interconnect comprising a number of ATM Port Interconnect Controller (APIC) chips [17]. APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps. However, TAO also runs on conventional real-time interconnects such as VME backplanes and multi-processor shared memory environments.

3 Using Patterns to Build an Extensible ORB Middleware

This section uses TAO as a case study to illustrate how patterns can help application developers and ORB developers build, maintain, and extend communication software by reducing the coupling between components. The patterns described in this section are not limited to ORBs or communication middleware, however. They have been applied in many other communication application domains, including telecom call processing and switching, avionics flight control systems, multimedia teleconferencing, and distributed interactive simulations.

Figure 3 illustrates the patterns used to develop an extensible ORB architecture for TAO. It is beyond the scope of this section to describe each pattern in detail or to discuss all the patterns used within TAO. Instead, our goal is to focus on the key patterns and show how they can improve the extensibility, maintainability, and performance of complex ORB middleware. The references point to additional material on each pattern.

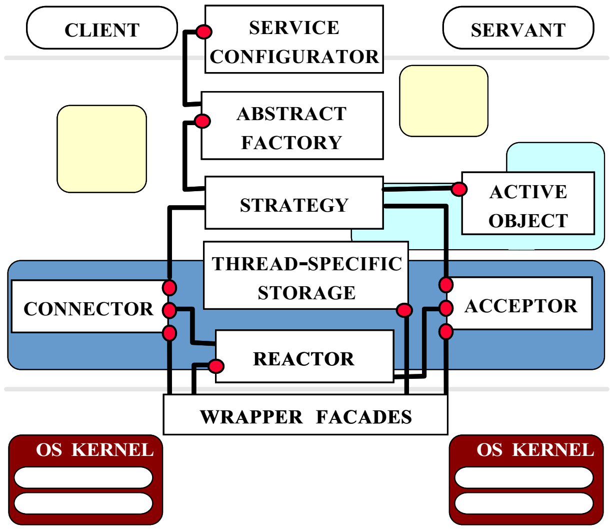

Figure 3: Relationships Among Patterns Used in TAO

In the following discussion, we outline the forces that underlie the main design challenges related to developing extensible and maintainable ORBs. We also explain how the absence of these patterns leaves these forces unresolved. In addition, we describe which patterns resolve these forces and illustrate how TAO implements these patterns to create an extensible and maintainable real-time ORB.

3.1 Encapsulate Low-level System Mechanisms with the Wrapper Facade Pattern

Context: One role of an ORB is to shield application-specific clients and servants from the details of low-level systems programming. Thus, ORB developers, rather than application developers, are responsible for tedious, low-level tasks like demultiplexing events, sending and receiving requests from the network, and spawning threads to execute requests concurrently.

Problem: Developing an ORB is difficult. It is even more difficult if developers must wrestle with low-level system APIs written in languages like C, which often causes the following problems:

- **ORB developers must have intimate knowledge of many OS platforms:** Implementing an ORB using system-level APIs forces developers to deal with the non-portable, tedious, and error-prone OS idiosyncrasies such as using untyped socket handles to identify connection endpoints. Moreover, these APIs are not portable across OS platforms. For example, Win32 lacks Pthreads and has subtly different semantics for sockets and `select`.

- **Increased maintenance effort:** One way to build an ORB is to handle portability variations via explicit conditional compilation directives in ORB source code. Using conditional compilation to address platform-specific variations *at all points of use* increases the complexity of the source code. It is hard to maintain and extend such ORBs, however, since platform-specific details are scattered throughout the implementation files.

- **Inconsistent programming paradigms:** System mechanisms are accessed through C-style function calls, which cause an "impedance mismatch" with the OO programming style supported by C++, the language used to implement TAO.

How can we avoid accessing low-level system mechanisms when implementing an ORB?

Solution → the Wrapper Facade pattern: An effective way to avoid accessing system mechanisms directly is to use the *Wrapper Facade pattern*. This pattern is a variant of the Facade pattern [8]. The intent of the Facade pattern is to simplify the interface for a subsystem. The intent of the Wrapper Facade pattern is more specific: it provides type-safe, modular, and portable class interfaces that encapsulate lower-level, stand-alone system mechanisms such as sockets, `select`, and Pthreads. In general, the Wrapper Facade pattern should be applied when existing system-level APIs are non-portable and non-type-safe.

Using the Wrapper Facade pattern in TAO: TAO accesses all system mechanisms via the wrapper facades provided by ACE [18]. ACE is an OO framework that implements core concurrency and distribution patterns for communication software. It provides reusable C++ wrapper facades and framework components that are targeted to developers of high-performance, real-time applications and services across a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems (like VxWorks, Chorus, and LynxOS).

Figure 4 illustrates how the ACE C++ wrapper facades improve TAO's robustness and portability by encapsulating and enhancing native OS concurrency, communication, memory management, event demultiplexing, and dynamic linking mechanisms with type-safe OO interfaces. The OO encapsulation provided by ACE alleviates the need for TAO to access the weakly-typed system mechanisms directly. Therefore, C++ compilers can detect type system violations at compile-time rather than at run-time. ACE wrapper facades use C++ features to eliminate performance penalties that would otherwise be incurred from its additional type-safety and layer of abstraction. For instance, inlining is used to avoid the overhead of calling small method calls. Likewise, static methods are used to avoid the overhead of passing a `this` pointer to each invocation.

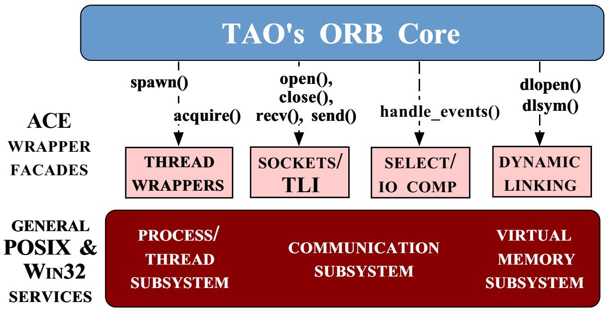

Figure 4: TAO's Wrapper Facade Encapsulation

Although the ACE wrapper facades solve a common development problem, they are just the first step towards developing an extensible and maintainable ORB. The remaining patterns described in this section build on the encapsulation provided by the ACE wrapper facades to address more challenging ORB design issues.

3.2 Demultiplexing ORB Core Events using the Reactor Pattern

Context: An ORB Core is responsible for demultiplexing I/O events from multiple clients and dispatching their associated event handlers. For instance, a server-side ORB Core listens for new client connections and reads/writes GIOP requests/responses from/to connected clients. To ensure responsiveness to multiple clients, an ORB Core uses OS event demultiplexing mechanisms to wait for CONNECTION, READ, and WRITE events to occur on *multiple* socket handles. Common event demultiplexing mechanisms include `select`, `WaitForMultipleObjects`, I/O completion ports, and threads.

Problem: One way to develop an ORB Core is to hard-code it to use one event demultiplexing mechanism. Relying on just one mechanism is undesirable, however, since no single scheme is efficient on all platforms or for all application requirements. For instance, asynchronous I/O completion ports are very efficient on Windows NT [19], whereas synchronous threads are the most efficient demultiplexing mechanism on Solaris [20].

Another way to develop an ORB Core is to tightly couple its event demultiplexing code with the code that performs GIOP protocol processing. In this case, the demultiplexing code cannot be reused as a blackbox component by similar communication middleware applications such as HTTP servers [19] or video-on-demand applications. Moreover, if new ORB strategies for threading or Object Adapter request scheduling algorithms are introduced, substantial portions of the ORB Core must be re-written.

How then can an ORB implementation render itself independent of a specific event demultiplexing mechanism and decouple its demultiplexing code from its handling code?

Solution → the Reactor pattern: An effective way to reduce coupling and increase the extensibility of an ORB Core is to apply the *Reactor pattern* [10]. This pattern supports synchronous demultiplexing and dispatching of multiple *event handlers*, which are triggered by events that can arrive concurrently from multiple sources. The Reactor pattern simplifies event-driven applications by integrating the demultiplexing of events and the dispatching of their corresponding event handlers. In general, the Reactor pattern should be applied when an application like an ORB Core must handle events from multiple clients concurrently, without commiting itself to a single low-level mechanism like `select`.

It is important to note that applying the Wrapper Facade pattern is not sufficient to resolve the problems outlined above. A wrapper facade for `select` may improve ORB Core portability somewhat. However, this pattern does not resolve the need to completely decouple the low-level event demultiplexing logic from the higher-level client request processing logic in an ORB Core.

Using the Reactor pattern in TAO: TAO uses the Reactor pattern to drive the main event loop within its ORB Core, as shown in Figure 5. A TAO server (**1**) initi-

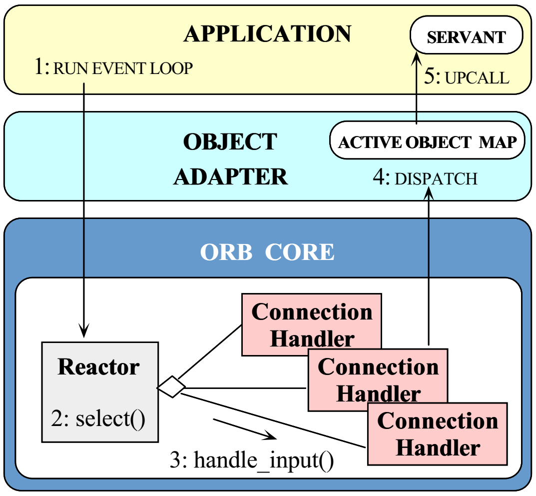

Figure 5: Using the Reactor Pattern in TAO's Event Loop

ates an event loop in the ORB Core's `Reactor`, where it (**2**) remains blocked on `select` until an I/O event occurs. When a GIOP request event occurs, the `Reactor` demultiplexes the request to the appropriate event handler, which is the GIOP `Connection_Handler` that is associated with each connected socket. The `Reactor` (**3**) then calls `Connection_Handler::handle_input`, which (**4**) dispatches the request to TAO's Object Adapter. The Object Adapter demultiplexes the request to the appropriate upcall method on the servant and (**5**) dispatches the upcall.

The Reactor pattern enhances the extensibility of TAO by decoupling the event handling portions of its ORB Core from the underlying OS event demultiplexing mechanisms. For example, the `WaitForMultipleObjects` event demultiplexing system call is used on Windows NT, whereas `select` is used on UNIX platforms. Moreover, the Reactor pattern simplifies the configuration of new event handlers. For instance, adding a new `Secure_Connection_Handler` that performs encryption/decryption of all traffic does not affect the Reactor's implementation.

3.3 Managing Connections in an ORB Using Acceptor-Connector Pattern

Context: Connection management is another key responsibility of an ORB Core. For instance, an ORB Core that implements the IIOP protocol must establish TCP connections and initialize the protocol handlers for each IIOP `server_endpoint`. By localizing connection management logic in the ORB Core, application-specific servants are able to focus solely on processing client requests.

An ORB Core is not *limited* to running over IIOP and TCP transports, however. For instance, while TCP can transfer GIOP requests reliably, its flow control and congestion control algorithms may preclude its use as a real-time protocol [13]. Likewise, it may be more efficient to use a shared memory transport mechanism when clients and servants are co-located on the same endsystem. Thus, an ideal ORB Core must be flexible in its support of multiple transport mechanisms.

Problem: The CORBA architecture explicitly decouples the connection management tasks performed by an ORB Core from the request processing performed by an application servant. However, one way to implement an ORB's *internal* connection management activities is to use low-level network APIs like sockets. Likewise, the connection establishment protocol can be tightly coupled with the communication protocol.

This design approach yields the following drawbacks, however:

1. Too inflexible: If an ORB's connection management data structures and algorithms are too closely intertwined, substantial effort is required to modify the ORB Core. For instance, tight coupling the ORB to use the socket API makes it hard to change the underlying transport mechanism, *e.g.*, to use shared memory rather than sockets. Therefore, it is intrusive and time consuming to port a tightly coupled ORB Core to new networks, such as ATM, or different network programming APIs, such as TLI or Win32 Named Pipes.

2. Too inefficient: Many internal ORB strategies can be optimized by allowing both ORB developers and application developers to select appropriate implementations late in the design cycle, *e.g.*, after extensive performance profiling. For example, a multi-threaded, real-time ORB client may need to store connection endpoints in thread-specific storage to reduce lock contention and overhead. Similarly, the concurrency strategy for a CORBA server might require that each connection run in its own thread to eliminate per-request locking overhead. However, if connection management mechanism are hard-coded and tightly bound with other internal ORB strategies it is hard to accommodate efficient new strategies.

How then can an ORB Core's connection management components support multiple transports and allow connection-related behaviors to be (re)configured flexibly late in the development cycle?

Solution → the Acceptor-Connector pattern: An effective way to increase the flexibility of ORB Core connection management and initialization is to apply the *Acceptor-Connector pattern* [21]. This pattern decouples connection initialization from the processing performed once a connection endpoint is initialized. The `Acceptor` component in the pattern is responsible for *passive* initialization, *i.e.*, the server-side of the ORB Core. Conversely, the `Connector` component in the pattern is responsible for *active* initialization, *i.e.*, the client-side of the ORB Core. In general, the Acceptor-Connector pattern should be applied when client/server middleware must allow flexible configuration of network programming APIs and must maintain proper separation of initialization roles.

Using the Acceptor-Connector pattern in TAO: TAO uses the Acceptor-Connector pattern in conjunction with the Reactor pattern to handle setup of connections for GIOP/IIOP communication. Within TAO's client-side ORB Core, a `Connector` initiates connections to servers in response to a method invocation or explicit binding to a remote object. Within TAO's server-side ORB Core, one or more `Acceptors` creates a GIOP `Connection Handler` to service each new client connection. `Acceptors` and `Connection_Handlers` both derive from `Event_Handler`, which enable them to be dispatched automatically by a `Reactor`.

TAO's `Acceptors` and `Connectors` can be configured with any transport mechanisms, such as sockets or TLI, provided by the ACE wrapper facades. In addition, TAO's `Acceptor` and `Connector` can be imbued with custom strategies to systematically select an appropriate concurrency mechanism, as described in Section 3.4.

Figure 6 illustrates the use of Acceptor-Connector strategies in TAO's ORB Core. When a client (**1**) invokes a

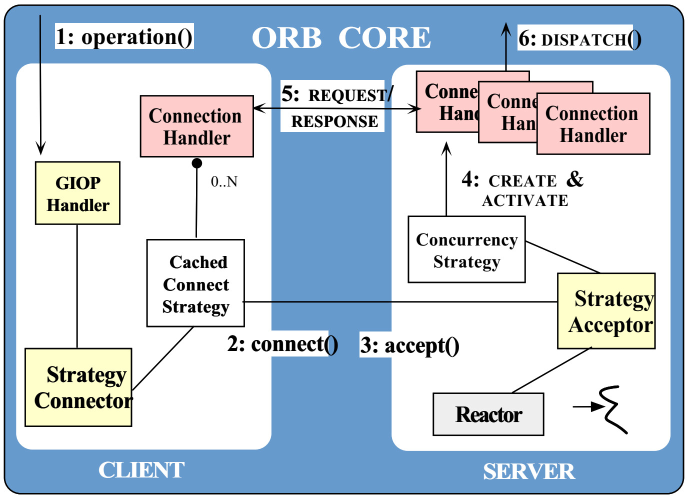

Figure 6: Using the Acceptor-Connector Pattern in TAO's Connection Management

remote operation, it makes a `connect` call through the `Strategy_Connector`. The `Strategy_Connector` (**2**) consults its connection strategy to obtain a connection. In this example the client uses a "caching connection strategy" that recycles connections to the server. Thus, it only creates new connections when all existing connections are already in use. This strategy minimizes connection setup time, thereby reducing end-to-end request latency.

In the server-side ORB Core, the `Reactor` notifies TAO's `Strategy_Acceptor` to (**3**) accept newly connected clients and create `Connection_Handlers`. The `Strategy_Acceptor` delegates the choice of concurrency mechanism to one of TAO's *concurrency* strategies (*e.g.*, reactive, thread-per-request, thread-per-connection, thread-per-priority, etc.) described in Section 3.4. Once a `Connection_Handler` is activated (**4**) within the ORB Core, it performs the requisite GIOP protocol processing (**5**) on a connection and ultimately dispatches (**6**) the request to the appropriate servant via TAO's Object Adapter.

3.4 Simplifying ORB Concurrency using the Active Object Pattern

Context: Once the Object Adapter has dispatched a client request to the appropriate servant, the servant executes the request. Execution may occur in the same thread of control as the `Connection Handler` that received it. Conversely, execution may occur in a different thread, concurrent with other request executions. The CORBA specification does not address the issue of concurrency within an ORB or a servant, leaving the decision to ORB developers and end-users.

It is important to develop ORBs that manage concurrent processing efficiently. Concurrency allows long-running operations to execute simultaneously without impeding the progress of other operations. Likewise, preemptive multi-threading is crucial to minimize the dispatch latency of real-time systems [14].

Problem: In many ORBs, the concurrency architecture is programmed directly using the OS platform's multi-threading API, such as the POSIX Pthreads API [22]. However, there are several drawbacks to this approach:

- **Non-portable:** Threading APIs tend to be very platform-specific. Even industry standards such as POSIX Pthreads are not available on many widely-used OS platforms, including Win32, VxWorks, and pSoS. Not only is there no direct mapping between APIs, but there is no clear mapping of functionality. For instance, POSIX Pthreads supports deferred thread cancellation, whereas Win32 threads do not. Moreover, although Win32 has a thread termination API, but the documentation strongly recommends *not* using it since it does not release thread resources on exit. Moreover, Pthreads itself is non-portable since many UNIX vendors implement different drafts of the standard.

- **Hard to program correctly:** Programming a multi-threaded ORB is hard since application and ORB developers must ensure that access to shared data is serialized properly in the ORB and in the servants. In addition, the techniques required to robustly terminate servants that execute concurrently in multiple threads are complicated, non-portable, and non-intuitive.

• **Non-extensible:** The choice of an ORB concurrency strategy depends largely on external factors like application requirements and network/endsystem characteristics. For instance, Reactive single-threading [10] is an appropriate strategy for short duration, compute-bound requests on a uni-processor. If these external factors change, however, it is important that an ORB can be extended to handle alternative concurrency strategies such as thread-per-request, thread pool, or thread-per-priority.

When ORBs are developed using low-level threading APIs, however, they are hard to extend it with new concurrency strategies *without* affecting other ORB components. How then can an ORB support a simple, extensible, and portable concurrency mechanism?

Solution → the Active Object pattern: An effective way to increase the portability, correctness, and extensibility of ORB concurrency strategies is to apply the *Active Object pattern* [11]. This pattern provides a higher-level concurrency architecture that decouples the thread that initially receives and processes a client request from the thread that ultimately executes the request.

While *Wrapper Facades* provide the basis for portability, they are simply thin veneers over the low-level system mechanisms. Moreover, a facade's behavior may still vary across platforms. Therefore, the Active Object pattern defines a higher-level concurrency abstraction that shields TAO from the complexity of low-level thread facades. By raising the level of abstraction for ORB developers, the Active Object pattern makes it easier to define more portable, flexible, and easy to program ORB concurrency strategies.

In general, the Active Object pattern should be used when an application can be simplified by centralizing the point where concurrency decisions are made. This pattern gives developers the flexibilityx to insert decision points between each request's initial reception and its ultimate execution. For instance, developers could decide whether or not to spawn a thread-per-connection or a thread-per-request.

Using the Active Object pattern in TAO: TAO uses the Active Object pattern to transparently allow a GIOP `Connection Handler` to execute requests either *reactively* by borrowing the Reactor's thread of control or *actively* by running in its own thread of control. The sequence of steps is shown in Figure 7.

The processing shown in Figure 7 is triggered when (**1**) a `Reactor` notifies the `Connection Handler` that an I/O event is pending. Based on the currently configured strategy, *e.g.*, reactive, thread-per-connection, thread-per-request,

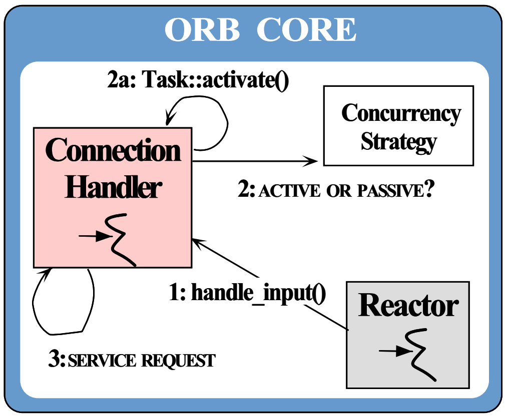

Figure 7: Using the Active Object Pattern to Structure TAO's Concurrency Strategies

etc., the handler (**2**) determines if it should be active or passive and acts accordingly. This flexibility is achieved by inheriting TAO's ORB Core connection handling classes from an ACE base class called `Task`. To process a request concurrently, therefore, the handler simply (**2a**) invokes the `Task::activate` method. This method spawns a new thread and invokes a standard hook method. Whether active or passive, the handler will ultimately (**3**) process the request.

3.5 Reducing Lock Contention and Priority Inversions with the Thread-Specific Storage Pattern

Context: The Active Object pattern allows applications and components in the ORB to operate using a variety of concurrency strategies, rather than one enforced by the ORB itself. The primary drawback to concurrency, however, is the need to serialize access to shared resources, such as operators `new` and `delete`, pointers created by the `CORBA::ORB_init` ORB initialization factory, or the `Acceptor` and `Connector` described in Section 3.3. A common way to achieve serialization is to use mutual-exclusion locks on each resource shared by multiple threads. However, acquiring and releasing these locks can be expensive, often negating any potential performance benefits of concurrency.

Problem: In theory, multi-threading an ORB can improve performance by executing multiple instruction streams simultaneously. In addition, multi-threading can simplify internal ORB design by allowing each thread to execute synchronously rather than reactively or asynchronously. In practice, multi-threaded ORBs often perform no better, or even worse, than single-threaded ORBs due to (1) the cost of acquiring/releasing locks and (2) priority inversions that arise when high- and low-priority threads contend for the same locks [23].

In addition, multi-threaded ORBs are hard to program due to complex concurrency control protocols required to avoid race conditions and deadlocks.

Solution → the Thread-Specific Storage pattern: An effective way to minimize the amount of locking required to serialize access to resources shared within an ORB is to use the *Thread-Specific Storage* pattern [24]. This pattern allows multiple threads in an ORB to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access.

Using the Thread-Specific Storage Pattern in TAO: TAO uses the Thread-Specific Storage pattern to minimize lock contention and priority inversion for real-time applications. Internally, each thread in the TAO stores its ORB Core and Object Adapter components, *e.g.*, `Reactor`, `Acceptor`, `Connector`, `POA`, in thread-specific storage. When a thread accesses any of these components, they are retrieved by using a `key` as an index into the thread's internal thread-specific state, as shown in Figure 8. Therefore, no additional locking is required to access ORB state.

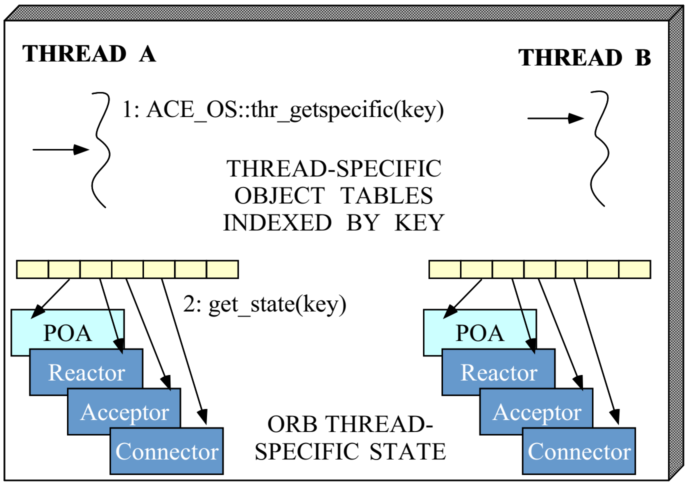

Figure 8: Using the Thread-Specific Storage Pattern TAO

3.6 Support Interchangeable ORB Behaviors with the Strategy Pattern

Context: The alternative concurrency architectures described in 3.4 are just one of the many strategies that an extensible ORB may be required to support. In general, extensible ORBs must support multiple request demultiplexing and scheduling strategies in their Object Adapters, as well as multiple connection establishment, request transfer, and concurrent request processing strategies in their ORB Cores.

Problem: One way to develop an ORB is to provide only static, non-extensible strategies, which are typically configured in the following ways:

- **Preprocessor macros:** Some strategies are determined by the value of preprocessor macros. For example, since threading is only available on selected platforms, conditional compilation is often used to select the appropriate concurrency architecture.

- **Command-line options:** Other strategies are controlled by the presence or absence of flags on the command-line. For instance, command-line options can be used to enable various ORB concurrency strategies.

While these two configuration approaches are widely used, they are very inflexible. For instance, preprocessor macros only support compile-time strategy selection, whereas command-line options convey a limited amount of information to an ORB. Moreover, these hard-coded configuration strategies are completely divorced from any code they might affect. Thus, ORB components that want to use these options must (1) know of their existence, (2) understand their range of values, and (3) provide an appropriate implementation for each value. These restrictions make it hard to develop highly extensible ORBs composed from transparently configurable strategies.

How then does an ORB (1) permit replacement of subsets of component strategies in a manner orthogonal and transparent to other ORB components and (2) encapsulate the state and behavior of each strategy so that changes to one component do not permeate throughout an ORB haphazardly?

Solution → the Strategy pattern: An effective way to support multiple transparently "pluggable" ORB strategies is to apply the *Strategy pattern* [8]. This pattern factors out similarity among algorithmic alternatives and explicitly associates the name of a strategy with its algorithm and state. Moreover, the Strategy pattern removes lexical dependencies on strategy implementations since applications access specialized behaviors only through common base class interfaces. In general, the Strategy pattern should be used when an application's behavior can be configured using multiple strategies that can be interchanged seamlessly.

Using the Strategy Pattern in TAO: TAO uses a variety of communication, concurrency, demultiplexing, real-time scheduling and dispatching strategies to factor out behaviors that are typically hard-coded in conventional ORBs. Several of these strategies are illustrated in Figure 9. For instance, TAO supports multiple request demultiplexing strategies (*e.g.*, perfect hashing vs. active demultiplexing [25]) and scheduling strategies (*i.e.*, FIFO vs. rate monotonic vs. earliest deadline first [14]) in its Object Adapter, as well as connection management strategies (*e.g.*, process-wide cached connections vs.

	Strategy			
Application	*Concurrency*	*Scheduling*	*Demultiplexing*	*Protocol*
Avionics	Thread-per-priority	Rate-based	Perfect hashing	VME backplane
Medical Imaging	Thread-per-connection	FIFO	Active demultiplexing	TCP/IP

Table 1: Example Applications and their ORB Strategy Configurations

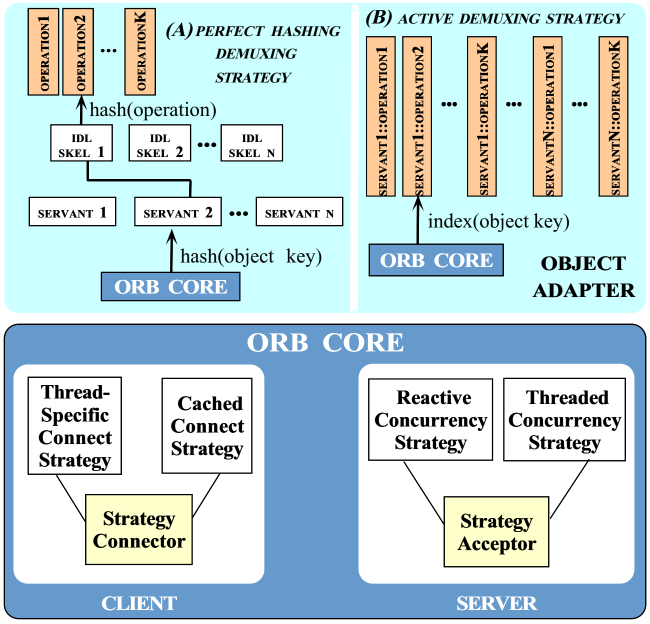

Figure 9: Strategies in TAO

thread-specific cached connections) and handler concurrency strategies (*e.g.*, Reactive vs. variations of Active Objects) in its ORB Core.

3.7 Consolidate ORB Strategies Using the Abstract Factory Pattern

Context: There are many potential strategy variants supported by TAO. Table 1 shows a simple example of the strategies used to create two configurations of TAO. One is an avionics application with hard real-time requirements [14] and the other is an electronic medical imaging application [15] with high throughput requirements. In general, the forces that must be resolved to compose all ORB strategies correctly are the need to (1) ensure the configuration of semantically compatible strategies and (2) simplify the management of a large number of individual strategies.

Problem: An undesirable side-effect of using the Strategy pattern extensively in complex software like ORBs is that extensibility becomes hard to manage for the following reasons:

- **Software complexity:** ORB source code can become littered with hard-coded references to strategy types. Many independent strategies must act in harmony to provide a comprehensive solution to particular application domains, such as real-time avionics. However, identifying these strategies individually by name requires tedious replacement of selected strategies in one domain with a potentially different set of strategies in another domain.

- **Semantic incompatibilities:** It is not always possible for certain ORB strategies to interact compatibly. For instance, the FIFO strategy for scheduling requests shown in Table 1 might not work with the thread-per-priority concurrency architecture. The problem stems from semantic incompatibilities between scheduling requests in their order of arrival, *i.e.*, FIFO queueing, versus dispatching requests based on their relative priorities, *i.e.*, preemptive priority-based thread dispatching. Moreover, some strategies are only useful when certain preconditions are met. For instance, the perfect hashing demultiplexing strategy is generally feasible only for systems that statically configure all servants off-line.

How can a highly-configurable ORB reduce the complexities required in managing its myriad of strategies, as well as enforce semantic consistency when combining discrete strategies?

Solution → the Abstract Factory pattern: An effective way to consolidate multiple ORB strategies into semantically compatible configurations is to apply the *Abstract Factory pattern* [8]. This pattern provides a single access point that integrates all strategies used to configure an ORB. Concrete subclasses then aggregate semantically compatible application-specific or domain-specific strategies, which can be replaced wholesale in semantically meaningful ways. In general, the Abstract Factory pattern should be used when an application needs to consolidate the configuration of many strategies, each having multiple variations.

Using the Abstract Factory pattern in TAO: All of TAO's ORB strategies are consolidated into two abstract factories implemented as Singletons [8]. One factory encapsulates client-specific strategies, while the factory shown in Figure 10 encapsulates server-specific strategies. These abstract factories encapsulate concurrency strategies in both the client and the

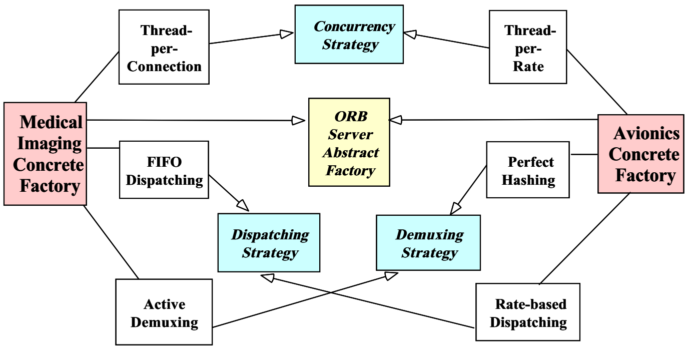

Figure 10: Factories used in TAO

server, and request demultiplexing, scheduling, and dispatch strategies in the server. By using the Abstract Factory pattern, TAO can configure different ORB personalities conveniently and consistently.

3.8 Dynamically Configure ORBs with the Service Configurator Pattern

Context: The cost of many computing resources, such as memory and CPUs, continue to drop. However, ORBs must still avoid excessive consumption of finite system resources. This parsimony is particularly essential for embedded real-time systems that require small memory footprints and predictable CPU processing overhead. Likewise, many applications can benefit from an ability to extend ORBs *dynamically* by allowing their strategies to be configured at run-time.

Problem: Although the Strategy and Abstract Factory patterns make it easier to customize ORBs for specific application requirements and system characteristics, these patterns can cause the following problems for extensible ORBs:

- **High resource utilization:** Widespread use of the Strategy pattern can substantially increase the number of strategies configured into an ORB, which can increase the system resources required to run an ORB.

- **Unavoidable system downtime:** If strategies are configured statically at compile-time or static link-time using abstract factories, it is hard to enhance existing strategies or add new strategies without (1) changing the existing source code for the consumer of the strategy or the abstract factory, (2) recompiling and relinking an ORB, and (3) restarting running ORBs and their application servants.

In general, static configuration is only feasible for a small, fixed number of strategies. Using this technique to configure complex ORBs complicates maintenance, increases system resource utilization, and leads to unavoidable system downtime to add or change existing components.

How then does an ORB implementation reduce the "overly-large, overly-static" side-effect of pervasive use of the Strategy and Abstract Factory patterns?

Solution $\rightarrow$ the Service Configurator pattern: An effective way to enhance the dynamism of an ORB is to apply the *Service Configurator pattern* [26]. This pattern uses explicit dynamic linking [27] mechanisms to obtain, utilize, and/or remove the run-time address bindings of custom Strategy and Abstract Factory objects into an ORB at installation-time or run-time. Widely available explicit dynamic linking mechanisms include the `dlopen/dlsym/dlclose` functions in SVR4 UNIX [28] and the `LoadLibrary/GetProcAddress` functions in the WIN32 subsystem of Windows NT [29]. The ACE wrapper facades provide a portable encapsulation of these OS functions.

By using the Service Configurator pattern, the *behavior* of ORB strategies are decoupled from *when* implementations of these strategies are configured into an ORB. For instance, ORB strategies can be linked into an ORB from DLLs at compile-time, installation-time, or even during run-time. Moreover, this pattern can reduce the memory footprint of an ORB by allowing application developers to dynamically link only those strategies that are necessary for a specific ORB personality.

In general, the Service Configurator pattern should be used when (1) an application wants to configure its constituent components dynamically and (2) conventional techniques, such as command-line options, are insufficient due to the number of possibilities or the inability to anticipate the range of values.

Using the Service Configurator pattern in TAO: TAO uses the Service Configurator pattern to configure abstract factories at run-time that contain the desired strategies. TAO's initialization code uses the dynamic linking mechanisms provided by the OS and encapsulated by the ACE wrapper facades to link in the appropriate factory for a particular use-case. This design allows applications to select the personality of a particular ORB at run-time. In addition, it allows the behavior of TAO to be tailored for specific platforms and application requirements without requiring access to the ORB source code.

Figure 11 shows two factories tuned for different application domains – avionics and medical imaging. In this particular configuration, the avionics concrete factory is currently installed in the process. Applications using this ORB configuration will be processed with a particular set of ORB concurrency, demultiplexing, and dispatching strategies. In contrast, the medical imaging concrete factory resides in a DLL outside

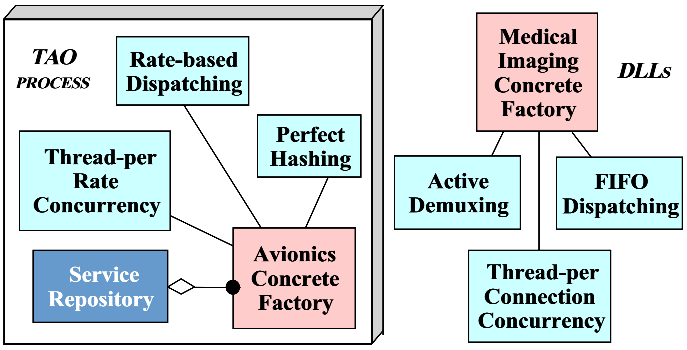

Figure 11: Using the Service Configurator Pattern in TAO

of the current ORB process. To support a different configuration of the ORB this factory could be dynamically installed when the ORB process is first initialized.

4 Concluding Remarks

This article presented a case study showing how we applied patterns to enhance the extensibility and maintainability of TAO, a dynamically configurable, real-time ORB. We found qualitative and quantitative evidence that the use of patterns helped to clarify the structure of, and collaboration between, components that perform key ORB tasks. These tasks include event demultiplexing and event handler dispatching, connection establishment and initialization of application services, concurrency control, and dynamic configuration. In addition, patterns improved TAO's performance and predictability by making it possible to transparently configure lightweight and optimized strategies for processing client requests.

In general, the use of patterns in TAO provided the following benefits:

• **Increased extensibility:** Patterns like Abstract Factory, Strategy, and Service Configurator make it much easier to reconfigure TAO for a particular application domain by allowing extensibility to be "designed into" an ORB. In contrast, middleware that lacks these patterns is significantly harder to develop and extend. This article illustrated how design patterns were applied to make an ORB more extensible.

• **Enhanced maintenance:** Design patterns are essential to capture and articulate the design rationale for complex structures in an ORB. Patterns help to demystify and motivate the structure of an ORB by describing its architecture in terms of design forces that recur in many software systems. The expressive power of patterns enabled us to convey the design of complex software systems like TAO.

Thus, the patterns presented in this article help to improve the maintainability of ORB middleware by reducing software complexity.

• **Increased portability and reuse:** Constructing our ORB atop an OO communication framework like ACE simplified the effort required to port TAO to various real-time platforms. Most of the porting effort is absorbed by the ACE framework maintainers. In addition, since the ACE framework is rich with configurable high-performance, real-time network-oriented components, we were able to achieve considerable code reuse by leveraging the framework.

The use of patterns can incur some liabilities. We summarize these liabilities below and discuss how we minimize them in TAO.

• **Abstraction penalty:** Many patterns use indirection to increase component decoupling. For instance, the Reactor pattern uses virtual methods to separate the application-specific `Event Handler` logic from the general-purpose event demultiplexing and dispatching logic. The extra indirection introduced by using these pattern implementations can potentially decrease performance. To alleviate these liabilities, we carefully applied C++ programming language features (such as inline functions and templates) and other optimizations (such as eliminating demarshaling overhead [30] and demultiplexing overhead [25]) to minimize performance overhead. As a result, TAO is substantially faster than the original hard-coded SunSoft IIOP [30].

• **Additional external dependencies:** Whereas SunSoft IIOP only depends on system-level interfaces and libraries, TAO now depends on the ACE framework. Since ACE encapsulates a wide range of low-level OS mechanisms, the effort required to port it to a new platform could potentially be higher than porting SunSoft IIOP, which only uses a subset of the OS's APIs. However, since ACE has been ported to many platforms already, the effort to port to new platforms is relatively low. Most sources of platform variation have been isolated to a few modules in ACE.

A final benefit of applying patterns to TAO is that not only did we developed a more flexible ORB, but we also devised a richer vocabulary for discussing ORB middleware designs. This vocabulary is a key "enabling" step to demystify the internals of an ORB. As we continue to learn about ORBs and the patterns of which they are composed, we expect this vocabulary to grow and evolve.

The source code for ACE and TAO is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

Acknowledgements

We would like to thank Frank Buschmann, Hans Rohnert, and Michael Stal for their extensive comments on this paper.

References

[1] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

[2] S. Landis and S. Maffeis, "Building Reliable Distributed Systems with CORBA," *Theory and Practice of Object Systems*, Apr. 1997.

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

[4] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

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

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

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

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

[9] D. C. Schmidt, "Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software," *Communications of the ACM (Special Issue on Object-Oriented Experiences)*, vol. 38, October 1995.

[10] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[11] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[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] 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.

[15] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[16] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston, "Real-Time CORBA," in *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, (Montréal, Canada), June 1997.

[17] Z. D. Dittia, G. M. Parulkar, and J. Jerome R. Cox, "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), IEEE, April 1997.

[18] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[19] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the 2^{nd} Global Internet Conference*, IEEE, November 1997.

[20] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[21] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[22] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[23] 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," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, (San Francisco, CA), IEEE, December 1997.

[24] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently," in *The 4^{th} Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[25] 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.

[26] 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.

[27] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[28] R. Gingell, M. Lee, X. Dang, and M. Weeks, "Shared Libraries in SunOS," in *Proceedings of the Summer 1987 USENIX Technical Conference*, (Phoenix, Arizona), 1987.

[29] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[30] A. Gokhale and D. C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance," in *Hawaiian International Conference on System Sciences*, January 1998.