Architectures and Patterns for Developing High-performance, Real-time ORB Endsystems

Douglas C. Schmidt, David L. Levine, and Chris Cleeland
{schmidt,levine,cleeland,irfan}@cs.wustl.edu

Department of Computer Science, Washington University
St. Louis, MO 63130[*]

September 20, 1998

This paper will appear in a chapter in the book series *Advances in Computers*, Academic Press, edited by Marvin Zelkowitz, to appear in 1999.

Abstract

Many types of applications can benefit from flexible and open middleware. CORBA is an emerging middleware standard for Object Request Brokers (ORBs) that simplifies the development of distributed applications and services. Experience with CORBA demonstrates that it is suitable for traditional RPC-style applications. However, the lack of performance optimizations and quality of service (QoS) features in conventional CORBA implementations make them unsuited for high-performance and real-time applications.

This paper makes four contributions to the design of CORBA ORBs for applications with high-performance and real-time requirements. First, it describes the design of TAO, which is our high-performance, real-time CORBA-compliant ORB. Second, it presents TAO's Real-time Scheduling Service, which provides QoS guarantees for deterministic real-time CORBA applications. Third, empirically evaluates the effects of priority inversion and non-determinism in conventional ORBs and shows how these hazards are avoided in TAO. Fourth, it presents a case study of key patterns used to develop TAO and quantifies the impact of applying patterns to reduce the complexity of common ORB tasks.

1 Introduction

Distributed computing helps improve application performance through multi-processing; reliability and availability through replication; scalability, extensibility, and portability through modularity; and cost effectiveness though resources sharing and open systems. An increasingly important class of distributed applications require stringent quality of service (QoS) guarantees. These applications include telecommunication systems command and control systems, multimedia systems, and simulations.

In addition to requiring QoS guarantees, distributed applications must be flexible and reusable. Flexibility is needed to respond rapidly to evolving functional and QoS requirements of distributed applications. Reusability is needed to yield substantial improvements in productivity and to enhance the quality, performance, reliability, and interoperability of distributed application software.

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. It simplifies 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 using low-level network programming mechanisms like sockets [4]. In particular, ORBs automate common network programming tasks such as object location, object activation, parameter marshaling/demarshaling, socket and request demultiplexing, fault recovery, and security. Thus, ORBs facilitate the development of flexible distributed applications and reusable services in heterogeneous distributed environments.

The remainder of this paper is organized as follows: Section 2 evaluates the suitability of CORBA for high-performance, real-time systems; Section 3 outlines the real-time feature enhancements and performance optimizations supported by TAO, which is our high-performance, real-time ORB endsystem; Section 4 describes the design of TAO's real-

[*]This work was supported in part by NSF grant NCR-9628218, DARPA contracts 9701516 and S30602-98-C-0187, Boeing, Lucent, Motorola, SAIC, Siemens, and Sprint.

time Scheduling Service; Section 5 qualitatively and quantitatively evaluates alternative ORB Core concurrency and connection architectures; Section 6 qualitatively and quantitatively evaluates the patterns that resolve key design challenges we faced when developing TAO; and Section 7 presents concluding remarks.

2 Evaluating OMG CORBA for High-performance, Real-time Systems

This section provides an overview of CORBA, explains why the current CORBA specification and conventional ORB implementations are currently inadequate for high-performance and real-time systems, and outlines the steps required to develop ORBs that can provide end-to-end QoS to applications.

2.1 Overview of the CORBA Reference Model

CORBA Object Request Brokers (ORBs) [1] allow clients to invoke operations on distributed objects without concern for the following issues [5]:

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.

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 2.x 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:

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 described below that ORBs use to transmit remote operation requests transparently from client to object.

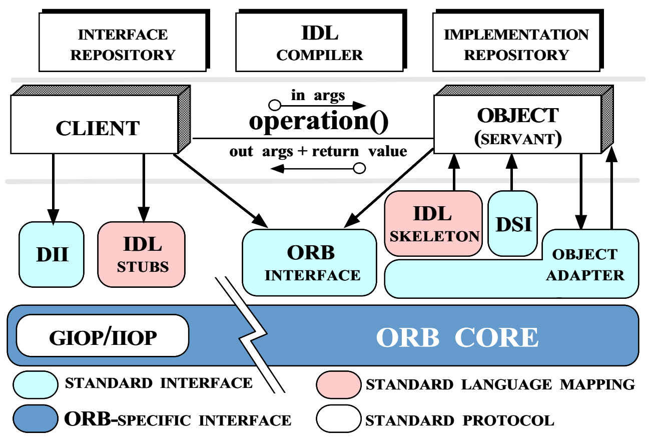

Figure 1: Components in the CORBA 2.x Reference Model

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. Over its lifetime, an object has one or more servants associated with it that implement its 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 class instances. 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 ORB Core communicates via a 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 [6].

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 accesses. 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, in CORBA 2.x, SII stubs only support *twoway*, *i.e.*, request/response, and *oneway*, *i.e.*, request-only operations.[1]

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 servants that have no compile-time knowledge of the IDL interface they implement. 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. CORBA 2.2 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 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 to CORBA objects, such as type libraries for stubs and skeletons.

Implementation Repository: The Implementation Repository [8] 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.

2.2 Limitations of CORBA for Real-time Applications

Our experience using CORBA on telecommunication [9], avionics [10], and medical imaging projects [11] indicates that it is well-suited for conventional RPC-style applications that possess "best-effort" quality of service (QoS) requirements. However, conventional CORBA implementations are not yet suited for high-performance, real-time applications for the following reasons:

Lack of QoS specification interfaces: The CORBA 2.x standard does not provide interfaces to specify end-to-end QoS requirements. For instance, there is no standard way for clients to indicate the relative priorities of their requests to an ORB. Likewise, there is no interface for clients to inform an ORB the rate at which to execute operations that have periodic processing deadlines.

The CORBA standard also does not define interfaces that allow applications to specify admission control policies. For instance, a video server might prefer to use available network bandwidth to serve a limited number of clients and refuse service to additional clients, rather than admit all clients and provide poor video quality [12]. Conversely, a stock quote service might want to admit a large number of clients and distribute all available bandwidth and processing time equally among them.

Lack of QoS enforcement: Conventional ORBs do not provide end-to-end QoS enforcement, *i.e.*, from application-to-application across a network. For instance, most ORBs transmit, schedule, and dispatch client requests in FIFO order. However, FIFO strategies can yield unbounded priority inversions [13, 14], which occur when a lower priority request blocks the execution of a higher priority request for an indefinite period. Likewise, conventional ORBs do not allow applications to specify the priority of threads that process requests.

Standard ORBs also do not provide fine-grained control of servant execution. For instance, they do not terminate servants

[1]The OMG has standardized an asynchronous method invocation interface in the Messaging specification [7], which will appear in CORBA 3.0.

that consume excess resources. Moreover, most ORBs use *ad hoc* resource allocation. Consequently, a single client can consume all available network bandwidth and a misbehaving servant can monopolize a server's CPU.

Lack of real-time programming features: The CORBA 2.x specification does not define key features that are necessary to support real-time programming. For instance, the CORBA General Inter-ORB Protocol (GIOP) supports asynchronous messaging. However, no standard programming language mapping exists in CORBA 2.x to transmit client requests asynchronously, though the Messaging specification in CORBA 3.0 will define this mapping. Likewise, the CORBA specification does not require an ORB to notify clients when transport layer flow control occurs, nor does it support timed operations [15]. As a result, it is hard to develop portable and efficient real-time applications that behave deterministically when ORB endsystem or network resources are unavailable temporarily.

Lack of performance optimizations: Conventional ORB endsystems incur significant throughput [11] and latency [16] overhead, as well as exhibiting many priority inversions and sources of non-determinism [17], as shown in Figure 2. These

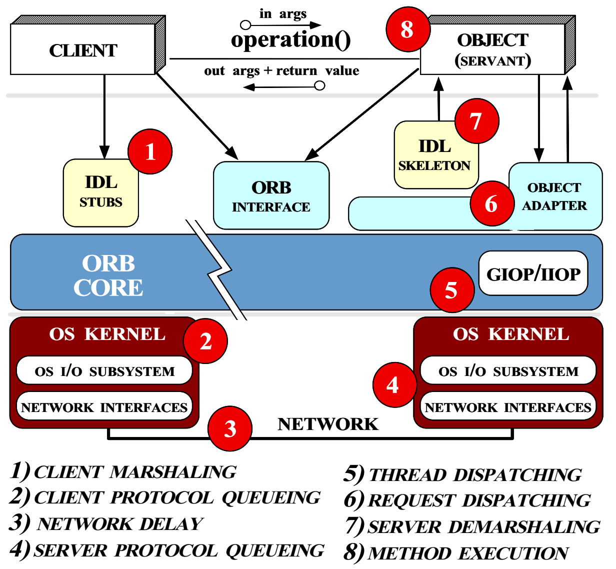

Figure 2: Sources of Latency and Priority Inversion in Conventional ORBs

overheads stem from (1) non-optimized presentation layers that copy and touch data excessively [6] and overflow processor caches [18]; (2) internal buffering strategies that produce non-uniform behavior for different message sizes [19]; (3) inefficient demultiplexing and dispatching algorithms [20]; (4) long chains of intra-ORB virtual method calls [21]; and (5) lack of integration with underlying real-time OS and network QoS mechanisms [22, 23, 17].

2.3 Overcoming CORBA Limitations for High-performance and Real-time Applications

Meeting the QoS needs of next-generation distributed applications requires much more than defining IDL interfaces or adding preemptive real-time scheduling to an OS. Instead, it requires a vertically and horizontally integrated *ORB endsystem* that can deliver end-to-end QoS guarantees at multiple levels throughout a distributed system. The key components in an ORB endsystem include the network interfaces, operating system I/O subsystems, communication protocols, and common middleware object services.

Implementing an effective framework for real-time CORBA requires ORB endsystem developers to address two types of issues: *QoS specification* and *QoS enforcement*. First, real-time applications must meet certain timing constraints to ensure the usefulness of the applications. For instance, a video-conferencing application may require an upper bound on the propagation delay of video packets from the source to the destination. Such constraints are defined by the *QoS specification* of the system. Thus, providing effective OO middleware requires a real-time ORB endsystem that supports the mechanisms and semantics for applications to specify their QoS requirements. Second, the architecture of the ORB endsystem must be designed carefully to *enforce* the QoS parameters specified by applications.

Section 3 describes how we are developing such an integrated middleware framework called *The ACE ORB* (TAO) [22]. TAO is a high-performance, real-time CORBA-compliant ORB endsystem developed using the ACE framework [24], which is a highly portable OO middleware communication framework. ACE contains a rich set of C++ components that implement strategic design patterns [25] for high-performance and real-time communication systems. Since TAO is based on ACE it runs on a wide range of OS platforms including general-purpose operating systems, such as Solaris and Windows NT, as well as real-time operating systems such as VxWorks, Chorus, and LynxOS.

2.3.1 Synopsis of TAO

The TAO project focuses on the following topics related to real-time CORBA and ORB endsystems:

- Identifying enhancements to standard ORB specifications, particularly OMG CORBA, that will enable applications to specify their QoS requirements concisely and precisely to ORB endsystems [26].
- Empirically determining the features required to build real-time ORB endsystems that can enforce deterministic and statistical end-to-end application QoS guarantees [23].

- Integrating the strategies for I/O subsystem architectures and optimizations [17] with ORB middleware to provide end-to-end bandwidth, latency, and reliability guarantees to distributed applications.
- Capturing and documenting the key design patterns [25] necessary to develop, maintain, configure, and extend real-time ORB endsystems.

In addition to providing a real-time ORB, TAO is an integrated ORB endsystem that consists of a high-performance I/O subsystem [27, 28] and an ATM Port Interconnect Controller (APIC) [29]. Figure 4 illustrates the main components in TAO's ORB endsystem architecture.

2.3.2 Requirements for High-performance and Real-time ORB Endsystems

The remainder of this section describes the requirements and features of ORB endsystems necessary to meet high-performance and real-time application QoS needs. It outlines key performance optimizations and provides a roadmap for the ORB features and optimizations presented in subsequent sections. Figure 3 summarizes the material covered below.

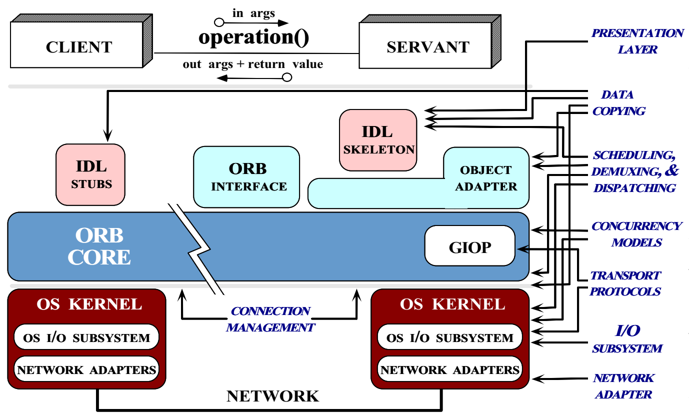

Figure 3: Features and Optimizations for Real-time ORB Endsystems

Policies and mechanisms for specifying end-to-end application QoS requirements: ORB endsystems must allow applications to specify the QoS requirements of their IDL operations using a small number of application-centric, rather than OS/network-centric parameters. Typical QoS parameters include computation time, execution period, and bandwidth/delay requirements. For instance, video-conferencing groupware [30, 12] may require high throughput and *statistical* real-time latency deadlines. In contrast, avionics mission control platforms [10] may require rate-based periodic processing with *deterministic* real-time deadlines.

QoS specification is not addressed by the CORBA 2.x specification, though there is an OMG special interest group (SIG) devoted to this topic. Section 4.3 explains how TAO allows applications to specify their QoS requirements using a combination of standard OMG IDL and QoS-aware ORB services.

QoS enforcement from real-time operating systems and networks: Regardless of the ability to *specify* application QoS requirements, an ORB endsystem cannot deliver end-to-end guarantees to applications without network and OS support for QoS *enforcement*. Therefore, ORB endsystems must be capable of scheduling resources such as CPUs, memory, and network connection bandwidth and latency. For instance, OS scheduling mechanisms must allow high-priority client requests to run to completion and prevent unbounded priority inversion.

Another OS requirement is preemptive dispatching. For example, a thread may become runnable that has a higher priority than one currently running a CORBA request on a CPU. In this case, the low-priority thread must be preempted by removing it from the CPU in favor of the high-priority thread.

Section 3.1 describes the OS I/O subsystem and network interface we are integrating with TAO. This infrastructure is designed to scale up to support performance-sensitive applications that require end-to-end gigabit data rates, predictable scheduling of I/O within an ORB endsystem, and low latency to CORBA applications.

Efficient and predictable real-time communication protocols and protocol engines: The throughput, latency, and reliability requirements of multimedia applications like teleconferencing are more stringent and diverse than those found in traditional applications like remote login or file transfer. Likewise, the channel speed, bit-error rates, and services (such as isochronous and bounded-latency delivery guarantees) of networks like ATM exceed those offered by traditional networks like Ethernet. Therefore, ORB endsystems must provide a protocol engine that is efficient, predictable, and flexible enough to be customized for different application QoS requirements and network/endsystem environments.

Section 3.2.1 outlines TAO's protocol engine, which provides real-time enhancements and high-performance optimizations to the standard CORBA General Inter-ORB Protocol (GIOP) [1]. The GIOP implementation in TAO's protocol engine specifies (1) a connection and concurrency architecture that minimizes priority inversion and (2) a transport protocol that enables efficient, predictable, and interoperable processing and communication among heterogeneous ORB endsystems.

Efficient and predictable request demultiplexing and dispatching: ORB endsystems must demultiplex and dispatch incoming client requests to the appropriate operation of the target servant. In conventional ORBs, demultiplexing occurs at multiple layers, including the network interface, the protocol stack, the user/kernel boundary, and several levels in an ORB's Object Adapter. 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 ORB endsystem. To minimize this overhead, and to ensure predictable dispatching behavior, TAO applies the perfect hashing and active demultiplexing optimizations [20] described in Section 3.3 to demultiplex requests in $O(1)$ time.

Efficient and predictable presentation layer: ORB presentation layer conversions transform application-level data into a portable format that masks byte order, alignment, and word length differences. Many performance optimizations have been designed to reduce the cost of presentation layer conversions. For instance, [31] describes the tradeoffs between using compiled vs. interpreted code for presentation layer conversions. Compiled marshaling code is efficient, but requires excessive amounts of memory. This can be problematic in many embedded real-time environments. In contrast, interpreted marshaling code is slower, but more compact and can often utilize processor caches more effectively.

Section 3.4 outlines how TAO supports predictable performance guarantees for both interpreted and compiled marshaling operations via its GIOP protocol engine. This protocol engine applies a number of innovative compiler techniques [6] and optimization principles [18]. These principles include optimizing for the common case; eliminating gratuitous waste; replacing general purpose operations with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; passing information between layers; and optimizing for the cache.

Efficient and predictable memory management: On modern high-speed hardware platforms, data copying consumes a significant amount of CPU, memory, and I/O bus resources [32]. Likewise, dynamic memory management incurs a significant performance penalty due to locking overhead and nondeterminism due to heap fragmentation. Minimizing data copying and dynamic memory allocation requires the collaboration of multiple layers in an ORB endsystem, *i.e.*, the network interfaces, I/O subsystem protocol stacks, ORB Core and Object Adapter, presentation layer, and application-specific servants.

Section 3.5 outlines TAO's vertically integrated memory management scheme that minimizes data copying and lock contention throughout its ORB endsystem.

2.3.3 Real-time vs. High-performance Tradeoffs

There is a common misconception [33] that applications with "real-time" requirements are equivalent to application with "high-performance" requirements. This is not necessarily the case. For instance, an Internet audio-conferencing system may not require high bandwidth, but it does require predictably low latency to provide adequate QoS to users in real-time.

Other multimedia applications, such as teleconferencing, have both real-time and high-performance requirements. Applications in other domains, such as avionics and process control, have stringent periodic processing deadline requirements in the worst-case. In these domains, achieving predictability in the worst-case is often more important than high performance in the average-case.

It is important to recognize that high-performance requirements may conflict with real-time requirements. For instance, real-time scheduling policies often rely on the predictability of endsystem operations like thread scheduling, demultiplexing, and message buffering. However, certain optimizations can improve performance at the expense of predictability. For instance, using a self-organizing search structure to demultiplex client requests in an ORB's Object Adapter can increase the average-case performance of operations, which decreases the predictability of any given operation in the worst-case.

To allow applications to select the appropriate tradeoffs between average-case and worst-case performance, TAO is designed with an extensible software architecture based on key communication patterns [25]. When appropriate, TAO employs algorithms and data structures that can optimize for both performance and predictability. For instance, the de-layered active demultiplexing scheme described in Section 3.3 can increase ORB performance *and* predictability by eliminating excessive searching and avoiding priority inversions across demultiplexing layers [20].

3 Architectural Components and Features for High-performance, Real-time ORB Endsystems

TAO's ORB endsystem contains the network interface, I/O subsystem, communication protocol, and CORBA middleware components shown in Figure 4. These components include the following.

1. I/O subsystem: which send/receives requests to/from clients in real-time across a network (such as ATM) or backplane (such as VME or compactPCI).

2. Run-time scheduler: which determines the priority at which requests are processed by clients and servers in an ORB endsystem.

3. ORB Core: which provides a highly flexible, portable, efficient, and predictable CORBA inter-ORB protocol engine

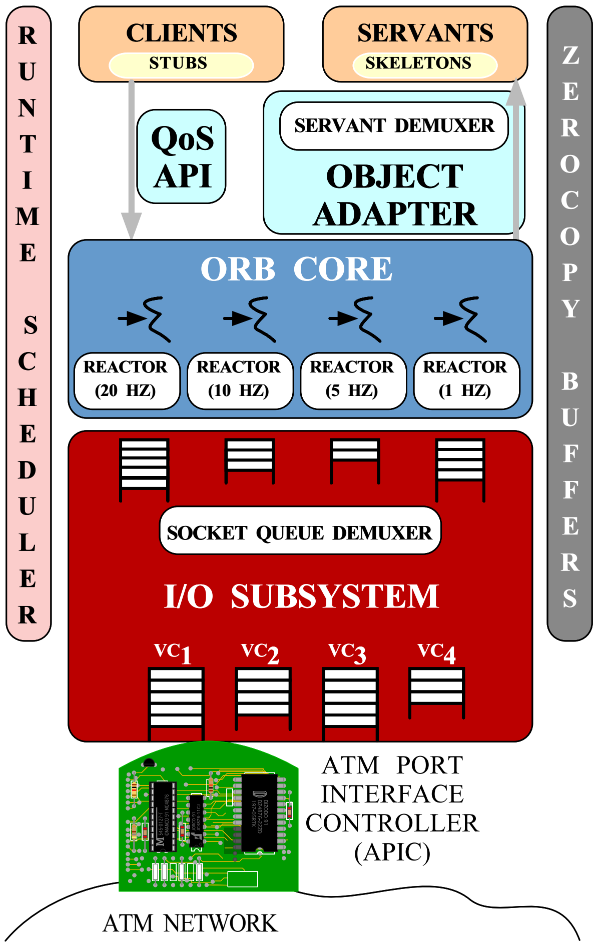

Figure 4: Architectural Components in the TAO Real-time ORB Endsystem

that delivers client requests to the Object Adapter and returns responses (if any) to clients.

4. Object Adapter: which demultiplexes and dispatches client requests optimally to servants using perfect hashing and active demultiplexing.

5. Stubs and skeletons: which optimize key sources of marshaling and demarshaling overhead in the code generated automatically by TAO's IDL compiler.

6. Memory manager: which minimizes sources of dynamic memory allocation and data copying throughout the ORB endsystem.

7. QoS API: which allows applications and higher-level CORBA services to specify their QoS parameters using an OO programming model.

TAO's I/O subsystem and portions of its run-time scheduler and memory manager run in the kernel. Conversely, TAO's ORB Core, Object Adapter, stubs/skeletons, and portions of its run-time scheduler and memory manager run in user-space.

The remainder of this section describes components 1, 3, 4, 5, and 6 and explains how they are implemented in TAO to meet the requirements of high-performance, real-time ORB endsystems described in Section 2.3. Section 4 focuses on components 2 and 7, which allow applications to specify QoS requirements for real-time servant operations. This paper discusses both high-performance and real-time features in TAO since it is designed to support applications with a wide range of QoS requirements.

3.1 High-performance, Real-time I/O Subsystem

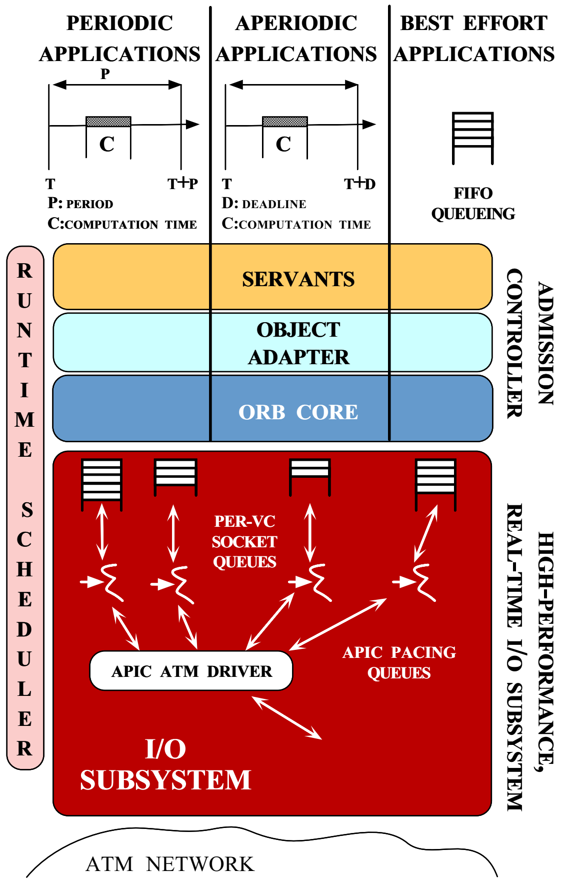

Figure 5: Components in TAO's High-performance, Real-time I/O Subsystem

An I/O subsystem is responsible for mediating ORB and application access to low-level network and OS resources such as device drivers, protocol stacks, and CPU(s). The key challenges in building a high-performance, real-time I/O subsystem are to (1) make it convenient for applications to specify their QoS requirements, (2) enforce QoS specifications and minimize priority inversion and non-determinism, and (3) enable ORB middleware to leverage QoS features provided by the underlying network and OS resources.

To meet these challenges, we have developed a high-performance, real-time network I/O subsystem that is customized for TAO [17]. The components in this subsystem are shown in Figure 5. They include (1) a high-speed ATM network interface, (2) a high-performance, real-time I/O subsystem, (3) a real-time Scheduling Service and Run-Time Scheduler, and (4) an admission controller, as described below.

High-speed network interface: At the bottom of TAO's I/O subsystem is a "daisy-chained" interconnect containing one or more ATM Port Interconnect Controller (APIC) chips [29]. APIC can be used both as an endsystem/network interface and as an I/O interface chip. It sustains an aggregate bi-directional data rate of 2.4 Gbps.

Although TAO is optimized for the APIC I/O subsystem, it is designed using a layered architecture that can run on conventional OS platforms, as well. For instance, TAO has been ported to real-time interconnects, such as VME and compact-PCI backplanes [17] and multi-processor shared memory environments, and QoS-enabled networks, such as IPv6 with RSVP [34].

Real-time I/O Subsystem: Some general-purpose operating systems like Solaris and Windows NT now support real-time scheduling. For example, Solaris 2.x provides a real-time scheduling class [14] that attempts to bound the time required to dispatch threads in this thread class. However, general-purpose operating systems do not provide real-time I/O subsystems. For instance, the Solaris STREAMS [35] implementation does not support QoS guarantees since STREAMS processing is performed at system thread priority, which is lower than all real-time threads [17]. Therefore, the Solaris I/O subsystem is prone to priority inversion since low-priority real-time threads can preempt the I/O operations of high-priority threads. Unbounded priority inversion is highly undesirable in many real-time environments.

TAO enhances the STREAMS model provided by Solaris and real-time operating systems like VxWorks and LynxOS. TAO's real-time I/O (RIO) subsystem minimizes priority inversion and hidden scheduling[2] that arise during protocol processing. TAO minimizes priority inversion by pre-allocating a pool of kernel threads dedicated to protocol processing. These kernel threads are co-scheduled with a pool of application threads. The kernel threads run at the same priority as the application threads, which prevents the real-time scheduling hazards outlined above.

To ensure predictable performance, the kernel threads belong to a *real-time I/O* scheduling class. This scheduling class uses rate monotonic scheduling (RMS) [36, 37] to support real-time applications with periodic processing behavior. Once a real-time I/O thread is admitted by the OS kernel, TAO's RIO subsystem is responsible for (1) computing its priority relative to other threads in the class and (2) dispatching the thread periodically so that its deadlines are met.

Real-time Scheduling Service and Run-Time Scheduler: The scheduling abstractions defined by real-time operating systems like VxWorks, LynxOS, and POSIX 1003.1c [38] implementations are relatively low-level. For instance, they require developers to map their high-level application QoS requirements into lower-level OS mechanisms, such as thread priorities and virtual circuit bandwidth/latency parameters. This manual mapping step is non-intuitive for many application developers, who prefer to design in terms of objects and operations on objects.

To allow applications to specify their scheduling requirements in a higher-level, more intuitive manner, TAO provides a Real-time Scheduling Service. This service is a CORBA object that is responsible for allocating system resources to meet the QoS needs of the applications that share the ORB endsystem.

Applications can use TAO's Real-time Scheduling Service to specify the processing requirements of their operations in terms of various parameters, such as computation time C, period P, or deadline D. If all operations can be scheduled, the Scheduling Service assigns a priority to each request. At run-time, these priority assignments are then used by TAO's Run-time Scheduler. The Run-time Scheduler maps client requests for particular servant operations into priorities that are understood by the local endsystem's OS thread dispatcher. The dispatcher then grants priorities to real-time I/O threads and performs preemption so that schedulability is enforced at run-time. Section 4.2 describe the Run-Time Scheduler and Real-time Scheduling Service in detail.

Admission Controller: To ensure that application QoS requirements can be met, TAO performs admission control for its real-time I/O scheduling class. Admission control allows the OS to either guarantee the specified computation time or to refuse to admit the thread. Admission control is useful for real-time systems with deterministic and/or statistical QoS requirements.

This paper focuses primarily on admission control for ORB endsystems. Admission control is also important at higher-levels in a distributed system, as well. For instance, admis-

[2]Hidden scheduling occurs when the kernel performs work asynchronously without regard to its priority. STREAMS processing in Solaris is an example of hidden scheduling since the computation time is not accounted for by the application or OS scheduler. To avoid hidden scheduling, the kernel should perform its work at the priority of the thread that requested the work.

sion control can be used for global resource managers [39, 40] that map applications onto computational, storage, and network resources in a large-scale distributed system, such as a ship-board computing environment.

3.2 Efficient and Predictable ORB Cores

The ORB Core is the component in the CORBA architecture that manages transport connections, delivers client requests to an Object Adapter, and returns responses (if any) to clients. The ORB Core typically implements the ORB's transport endpoint demultiplexing and concurrency model, as well.

The key challenges to developing a real-time ORB Core are (1) implementing an efficient protocol engine for CORBA inter-ORB protocols like GIOP and IIOP, (2) determining a suitable connection and concurrency model that can share the aggregate processing capacity of ORB endsystem components predictably among operations in one or more threads of control, and (3) designing an ORB Core that can be adapted easily to new endsystem/network environments and application QoS requirements. The following describes how TAO's ORB Core is designed to meet these challenges.

3.2.1 TAO's Inter-ORB Protocol Engine

TAO's protocol engine is a highly optimized, real-time version of the SunSoft IIOP reference implementation [18] that is integrated with the high-performance I/O subsystem described in Section 3.1. Thus, TAO's ORB Core on the client, server, and any intermediate nodes can collaborate to process requests in accordance with their QoS attributes. This design allows clients to indicate the relative priorities of their requests and allows TAO to enforce client QoS requirements end-to-end.

To increase portability across OS/network platforms, TAO's protocol engine is designed as a separate layer in TAO's ORB Core. Therefore, it can either be tightly integrated with the high-performance, real-time I/O subsystem described in Section 3.1 or run on conventional embedded platforms linked together via interconnects like VME or shared memory.

Below, we outline the existing CORBA interoperability protocols and describe how TAO implements these protocols in an efficient and predictable manner.

Overview of GIOP and IIOP: CORBA is designed to run over multiple transport protocols. The standard ORB interoperability protocol is known as the General Inter-ORB Protocol (GIOP) [1]. GIOP provides a standard end-to-end interoperability protocol between potentially heterogeneous ORBs. GIOP specifies an abstract interface that can be mapped onto transport protocols that meet certain requirements, *i.e.*, connection-oriented, reliable message delivery, and untyped bytestream. An ORB supports GIOP if applications can use the ORB to send and receive standard GIOP messages.

The GIOP specification consists of the following elements:

• **Common Data Representation (CDR) definition:** The GIOP specification defines a common data representation (CDR). CDR is a transfer syntax that maps OMG IDL types from the native endsystem format to a bi-canonical format, which supports both little-endian and big-endian binary data formats. Data is transferred over the network in CDR encodings.

• **GIOP Message Formats:** The GIOP specification defines messages for sending requests, receiving replies, locating objects, and managing communication channels.

• **GIOP Transport Assumptions:** The GIOP specification describes what types of transport protocols can carry GIOP messages. In addition, the GIOP specification describes how connections are managed and defines constraints on message ordering.

The CORBA Inter-ORB Protocol (IIOP) is a mapping of GIOP onto the TCP/IP protocols. ORBs that use IIOP are able to communicate with other ORBs that publish their locations in an *interoperable object reference* (IOR) format.

Implementing GIOP/IIOP efficiently and predictably: In Corba 2.x, neither GIOP nor IIOP provide support for specifying or enforcing the end-to-end QoS requirements of applications.[3] This makes GIOP/IIOP unsuitable for real-time applications that cannot tolerate the latency overhead and jitter of TCP/IP transport protocols. For instance, TCP functionality like adaptive retransmissions, deferred transmissions, and delayed acknowledgments can cause excessive overhead and latency for real-time applications. Likewise, routing protocols like IPv4 lack functionality like packet admission policies and rate control, which can lead to excessive congestion and missed deadlines in networks and endsystems.

To address these shortcomings, TAO's ORB Core supports a priority-based concurrency architecture, a priority-based connection architecture, and a real-time inter-ORB protocol (RIOP), as described below.

• **TAO's priority-based concurrency architecture:** TAO's ORB Core can be configured to allocate a real-time thread[4] for each application-designated priority level. Every thread in TAO's ORB Core can be associated with a `Reactor`, which implements the Reactor pattern [43] to provide flexible and efficient endpoint demultiplexing and event handler dispatching.

[3]The forthcoming real-time CORBA specification [41] will support this capability.

[4]In addition, TAO's ORB Core can be configured to support thread pool, thread-per-connection, and single-threaded reactive dispatching [42].

When playing the role of a server, TAO's `Reactor`(s) demultiplex incoming client requests to connection handlers that perform GIOP processing. These handlers collaborate with TAO's Object Adapter to dispatch requests to application-level servant operations. Operations can either execute at (1) the priority of the client that invoked the operation or (2) at the priority of the real-time ORB Core thread that received the operation. The latter design is well-suited for deterministic real-time applications since it minimizes priority inversion and non-determinism in TAO's ORB Core [44]. In addition, it reduces context switching and synchronization overhead since servant state must be locked only if servants interact across different thread priorities.

TAO's priority-based concurrency architecture is optimized for statically configured, fixed priority real-time applications. In addition, it is well suited for scheduling and analysis techniques that associate priority with *rate*, such as rate monotonic scheduling (RMS) and rate monotonic analysis (RMA) [36, 37]. For instance, avionic mission computing systems commonly execute their tasks in *rates groups*. A rate group assembles all periodic processing operations that occur at particular rates (*e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz) and assigns them to a pool of threads using fixed-priority scheduling.

• **TAO's priority-based connection architecture:** Figure 6 illustrates how TAO can be configured with a priority-based connection architecture. In this model, each client

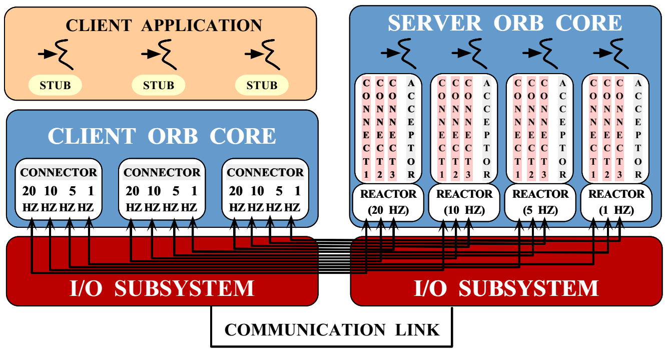

Figure 6: TAO's Priority-based Connection and Concurrency Architectures

thread maintains a `Connector` [45] in thread-specific storage. Each `Connector` manages a map of pre-established connections to servers. A separate connection is maintained for each thread priority in the server ORB. This design enables clients to preserve end-to-end priorities as requests traverse through ORB endsystems and communication links [44].

Figure 6 also shows how the `Reactor` in each thread priority in a server ORB can be configured to use an `Acceptor` [45]. The `Acceptor` is a socket endpoint factory that listens on a specific port number for clients to connect to the ORB instance running at a particular thread priority. TAO can be configured so that each priority level has its own `Acceptor` port. For instance, in statically scheduled, rate-based avionics mission computing systems [46], ports 10020, 10010, 10005, 10001 could be mapped to the 20 Hz, 10 Hz, 5 Hz, and 1 Hz rate groups, respectively. Requests arriving at these socket ports can then be processed by the appropriate fixed-priority real-time threads.

Once a client connects, the `Acceptor` in the server ORB creates a new socket queue and a GIOP connection handler to service that queue. TAO's I/O subsystem uses the port number contained in arriving requests as a demultiplexing key to associate requests with the appropriate socket queue. This design minimizes priority inversion through the ORB endsystem via *early demultiplexing* [27, 28, 29], which associates requests arriving on network interfaces with the appropriate real-time thread that services the target servant. As described in Section 8, early demultiplexing is used in TAO to vertically integrate the ORB endsystem's QoS support from the network interface up to the application servants.

• **TAO's Real-time inter-ORB protocol (RIOP):** TAO's connection-per-priority scheme described above is optimized for fixed-priority applications that transfer their requests at particular rates through statically allocated connections serviced at the priority of real-time server threads. Applications that possess dynamic QoS characteristics, or that propagate the priority of a client to the server, require a more flexible protocol, however. Therefore, TAO supports a real-time Inter-ORB Protocol (RIOP).

RIOP is an implementation of GIOP that allows ORB endsystems to transfer their QoS attributes end-to-end from clients to servants. For instance, TAO's RIOP mapping can transfer the *importance* of an operation end-to-end with each GIOP message. The receiving ORB endsystem uses this QoS attribute to set the priority of a thread that processes an operation in the server.

To maintain compatibility with existing IIOP-based ORBs, TAO's RIOP protocol implementation transfers QoS information in the `service_context` member of the `GIOP::requestHeader`. ORBs that do not support TAO's RIOP extensions can transparently ignore the `service_context` member. Incidentally, the RIOP feature will be standardized as a QoS property in the asynchronous messaging portion of the CORBA 3.0 specification.

The TAO RIOP `service_context` passed with each client invocation contains attributes that describe the operation's QoS parameters. Attributes supported by TAO's RIOP extensions include priority, execution period, and communication class. Communication classes supported by TAO include ISOCHRONOUS for continuous media, BURST for bulk data, MESSAGE for small messages with low delay requirements, and MESSAGE_STREAM for message sequences that must be processed at a certain rate [28].

In addition to transporting client QoS attributes, TAO's RIOP is designed to map CORBA GIOP on a variety of networks including high-speed networks like ATM LANs and ATM/IP WANs [47]. RIOP also can be customized for specific application requirements. To support applications that do not require complete reliability, TAO's RIOP mapping can selectively omit transport layer functionality and run directly atop ATM virtual circuits. For instance, teleconferencing or certain types of imaging may not require retransmissions or bit-level error detection.

3.2.2 Enhancing the Extensibility and Portability of TAO's ORB Core

Although most conventional ORBs interoperate via IIOP over TCP/IP, an ORB is not limited to running over these transports. 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. Likewise, shared memory may be a more effective transport mechanism when clients and servants are co-located on the same endsystem. Therefore, a key design challenge is to make an ORB Core extensible and portable to multiple transport mechanisms and OS platforms.

To increase extensibility and portability, TAO's ORB Core is based on patterns in the ACE framework [24]. Section 6 describes the patterns used in TAO in detail. The following outlines the patterns that are used in TAO's ORB Core.

TAO's ORB Core uses the *Strategy* and *Abstract Factory* patterns [48] to allow the configuration of multiple scheduling algorithms, such as earliest deadline first or maximum urgency first [49]. Likewise, the *Bridge* pattern [48] shields TAO's ORB Core from the choice of scheduling algorithm. TAO uses ACE components based on the *Service Configurator* pattern [50] to allow new algorithms for scheduling, demultiplexing, concurrency, and dispatching to be configured dynamically, *i.e.*, at runtime. On platforms with C++ compilers that optimize virtual function calls, the overhead of this extensibility is negligible [10].

Other patterns are used in TAO's ORB Core to simplify its connection and concurrency architectures. For instance, the *Acceptor-Connector* pattern [45] defines ACE components used in TAO to decouple the task of connection establishment from the GIOP processing tasks performed after connection establishment. TAO uses the *Reactor* pattern [43], which defines an ACE component that simplifies the event-driven portions of the ORB core by integrating socket demultiplexing and the dispatching of the corresponding GIOP connection handlers. Likewise, the *Active Object* pattern [51] defines an ACE component used in TAO to configure multiple concurrency architectures by decoupling operation invocation from operation execution.

TAO ports easily to many OS platforms since it is built using ACE components based on the patterns described above. Currently, ACE and TAO have been ported to a wide range of OS platforms including Win32 (*i.e.*, WinNT 3.5.x/4.x, Win95, and WinCE), most versions of UNIX (*e.g.*, SunOS 4.x and 5.x, SGI IRIX 5.x and 6.x, HP-UX 9.x, 10.x, and 11.x, DEC UNIX 4.x, AIX 4.x, Linux, SCO, UnixWare, NetBSD, and FreeBSD), real-time operating systems (*e.g.*, VxWorks, Chorus, LynxOS, and pSoS), and MVS OpenEdition.

Figure 7 illustrates the components in the client-side and server-side of TAO's ORB Core. The client-side uses a `Strategy_Connector` to create and cache `Connection_Handlers` that are bound to each server. These connections can be pre-allocated during ORB initialization. Pre-allocation minimizes the latency between client invocation and servant operation execution since connections can be established *a priori* using TAO's explicit binding operation.

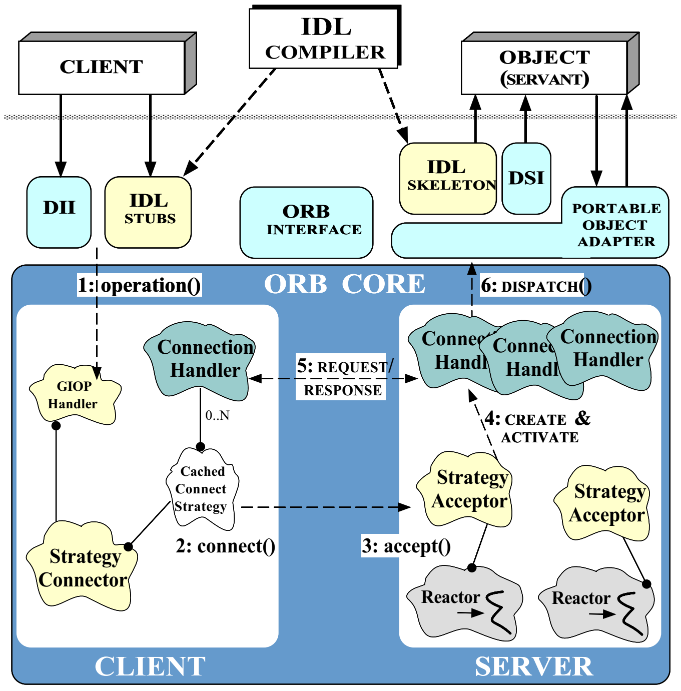

Figure 7: Components in the TAO's ORB Core

On the server-side, the `Reactor` detects new incoming connections and notifies the `Strategy_Acceptor`. The `Strategy_Acceptor` accepts the new connection and associates it with a `Connection_Handler` that executes in a thread with an appropriate real-time priority. The client's `Connection_Handler` can pass GIOP requests (described in Section 3.2.1) to the server's `Connection_Handler`.

This handler upcalls TAO's Object Adapter, which dispatches the requests to the appropriate servant operation.

3.2.3 Real-time Scheduling and Dispatching of Client Requests

TAO's ORB Core can be configured to implement custom mechanisms that process client requests according to application-specific real-time scheduling policies. To provide a guaranteed share of the CPU among application operations [28, 10], TAO's ORB Core uses the real-time Scheduling Service described in Section 4. One of the strategies provided by TAO's ORB Core is variant of periodic rate monotonic scheduling implemented with real-time threads and real-time upcalls (RTUs) [28].

TAO's ORB Core contains an object reference to its Run-Time Scheduler shown in Figure 4. This scheduler dispatches client requests in accordance with a real-time scheduling policy configured into the ORB endsystem. The Run-Time Scheduler maps client requests to real-time thread priorities and connectors.

TAO's initial implementation supports deterministic real-time applications [17]. In this case, TAO's Run-Time Scheduler consults a table of request priorities generated off-line. At run-time, TAO's ORB Core dispatches threads to the CPU(s) according to its dispatching mechanism. We are have extended TAO to support dynamically scheduling and applications with statistical QoS requirements [46].

3.3 Efficient and Predictable Object Adapters

The Object Adapter is the component in the CORBA architecture that associates a servant with an ORB, demultiplexes incoming client requests to the servant, and dispatches the appropriate operation of that servant. The key challenges associated with designing an Object Adapter for real-time ORBs are determining how to demultiplex client requests efficiently, scalably, and predictably.

TAO is the first CORBA ORB whose Object Adapter implements the OMG POA (Portable Object Adapter) specification [1]. The POA specification defines a wide range of features, including: user- or system-supplied Object Ids, persistent and transient objects, explicit and on-demand activation, multiple servant $\rightarrow$ CORBA object mappings, total application control over object behavior and existence, and static and DSI servants [52, 53].

The demultiplexing and dispatching policies in TAO's Object Adapter are instrumental to ensuring its predictability and efficiency. This subsection describes how TAO's Object Adapter can be configured to use perfect hashing or active demultiplexing to map client requests directly to servant/operation tuples in $O(1)$ time.

3.3.1 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 8. These steps perform the

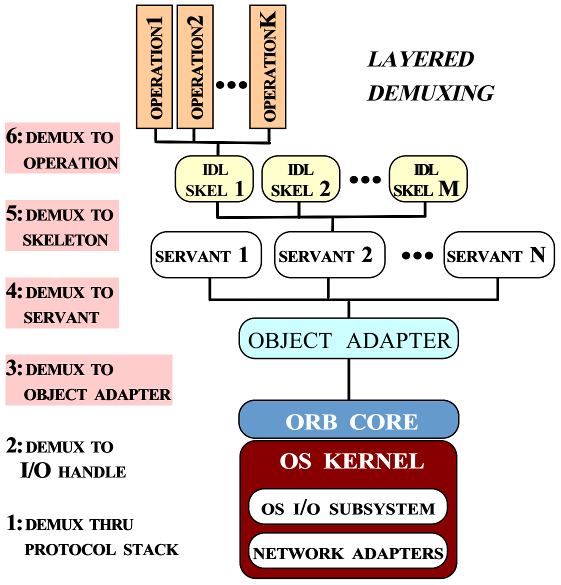

Figure 8: Layered CORBA Request Demultiplexing

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 [54]:

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 [17].

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [21, 16] show that conventional ORBs spend ~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.

3.3.2 TAO's Optimized ORB Demultiplexing Strategies

To address the limitations with conventional ORBs, TAO provides the demultiplexing strategies shown in Figure 9. TAO's optimized demultiplexing strategies include the following:

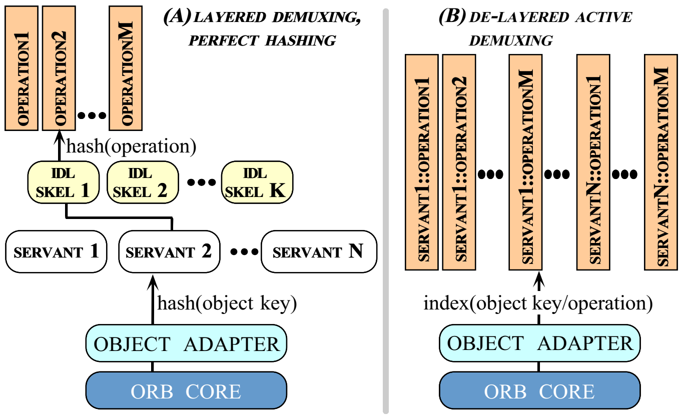

Figure 9: Optimized CORBA Request Demultiplexing Strategies

Perfect hashing: The perfect hashing strategy shown in Figure 9(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` [55] 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 [10, 46], 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 9(B). In this strategy, the client passes an object key that directly identifies the servant and operation in $O(1)$ time in the worst-case. The client obtains this object key 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 object key 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 [50] to select perfect hashing or active demultiplexing dynamically during ORB installation [25]. Both strategies improve request demultiplexing performance and predictability *above* the ORB Core.

To improve efficiency and predictability *below* the ORB Core, TAO uses the ATM Port Interconnect Controller (APIC) described in Section 3.1 to directly dispatch client requests associated with ATM virtual circuits [17]. This vertically integrated, optimized ORB endsystem architecture reduces demultiplexing latency and supports end-to-end QoS on either a per-request or per-connection basis.

3.4 Efficient and Predictable Stubs and Skeletons

Stubs and skeletons are the components in the CORBA architecture responsible for transforming typed operation parameters from higher-level representations to lower-level representations (marshaling) and vice versa (demarshaling). Marshaling and demarshaling are major bottlenecks in high-performance communication subsystems [56] due to the significant amount of CPU, memory, and I/O bus resources they consume while accessing and copying data. Therefore, key challenges for a high-performance, real-time ORB are to design an efficient presentation layer that performs marshaling and demarshaling predictably, while minimizing the use of costly operations like dynamic memory allocation and data copying.

In TAO, presentation layer processing is performed by client-side stubs and server-side skeletons that are generated automatically by a highly-optimizing IDL compiler [6]. In addition to reducing the potential for inconsistencies between client stubs and server skeletons, TAO's IDL compiler supports the following optimizations:

Reduced use of dynamic memory: TAO's IDL compiler analyzes the storage requirements for all the messages exchanged between the client and the server. This enables the compiler to allocate sufficient storage *a priori* to avoid repeated run-time tests that determine if sufficient storage is available. In addition, the IDL compiler uses the run-time stack to allocate storage for unmarshaled parameters.

Reduced data copying: TAO's IDL compiler analyzes when it is possible to perform block copies for atomic data types rather than copying them individually. This reduces excessive data access since it minimizes the number of load and store instructions.

Reduced function call overhead: TAO's IDL compiler can selectively optimize small stubs via *inlining*, thereby reducing the overhead of function calls that would otherwise be incurred by invoking these small stubs.

TAO's IDL compiler supports multiple strategies for marshaling and demarshaling IDL types. For instance, TAO's IDL compiler can generate either compiled and/or interpreted IDL stubs and skeletons. This design allows applications to select between (1) *interpreted* stubs/skeletons, which can be somewhat slower, but more compact in size and (2) *compiled* stubs/skeletons, which can be faster, but larger in size [31].

Likewise, TAO can cache premarshaled application data units (ADUs) that are used repeatedly. Caching improves performance when ADUs are transferred sequentially in "request chains" and each ADU varies only slightly from one transmission to the other. In such cases, it is not necessary to marshal the entire request every time. This optimization requires that the real-time ORB perform flow analysis [57, 58] of application code to determine what request fields can be cached.

Although these techniques can significantly reduce marshaling overhead for the common case, applications with strict real-time service requirements often consider only worst-case execution. As a result, the flow analysis optimizations described above can only be employed under certain circumstances, *e.g.*, for applications that can accept statistical real-time service or when the worst-case scenarios are still sufficient to meet deadlines.

3.5 Efficient and Predictable Memory Management

Conventional ORB endsystems suffer from excessive dynamic memory management and data copying overhead [21]. For instance, many I/O subsystems and ORB Cores allocate a memory buffer for each incoming client request and the I/O subsystem typically copies its buffer to the buffer allocated by the ORB Core. In addition, standard GIOP/IIOP demarshaling code allocates memory to hold the decoded request parameters. Likewise, IDL skeletons dynamically allocate and delete copies of client request parameters before and after upcalls, respectively.

In general, dynamic memory management is problematic for real-time systems. For instance, heap fragmentation can yield non-uniform behavior for different message sizes and different workloads. Likewise, in multi-threaded ORBs, the locks required to protect the heap from race conditions increase the potential for priority inversion [44]. In general, excessive data copying throughout an ORB endsystem can significantly lower throughput and increase latency and jitter.

TAO is designed to minimize and eliminate data copying at multiple layers in its ORB endsystem. For instance, TAO's buffer management system uses the APIC network interface to enhance conventional operating systems with a *zero-copy* buffer management system [29]. At the device level, the APIC interacts directly with the main system bus and other I/O devices. Therefore, it can transfer client requests between endsystem buffer pools and ATM virtual circuits with no additional data copying.

The APIC buffer pools for I/O devices described in Section 3.1 can be configured to support *early demultiplexing* of periodic and aperiodic client requests into memory shared among user- and kernel-resident threads. These APIs allow client requests to be sent/received to/from the network without incurring any data copying overhead. Moreover, these buffers can be preallocated and passed between various processing stages in the ORB, thereby minimizing costly dynamic memory management.

In addition, TAO uses the Thread-Specific Storage pattern [59] to minimize lock contention resulting from memory allocation. TAO can be configured to allocate its memory from thread-specific storage. In this case, when the ORB requires memory it is retrieved from a thread-specific heap. Thus, no locks are required for the ORB to dynamically allocate this memory.

4 Supporting Real-time Scheduling in CORBA

Section 3 described the architectural components used in TAO to provide a high-performance ORB endsystem for real-time CORBA. TAO's architecture has been realized with minimal changes to CORBA. However, the CORBA 2.x specification does not yet address issues related to real-time scheduling. Therefore, this section provides in-depth coverage of the components TAO uses to implement a Real-time Scheduling Service, based on standard CORBA features.

4.1 Synopsis of Application Quality of Service Requirements

The TAO ORB endsystem [23] is designed to support various classes of quality of service (QoS) requirements, including applications with deterministic and statistical real-time requirements. Deterministic real-time applications, such as avionics mission computing systems [10], must meet periodic deadlines. These types of applications commonly use static scheduling and analysis techniques, such as rate monotonic analysis (RMA) and rate monotonic scheduling (RMS).

Statistical real-time applications, such as teleconferencing and video-on-demand, can tolerate minor fluctuations in scheduling and reliability guarantees, but nonetheless require QoS guarantees. These types of applications commonly use dynamic scheduling techniques [46], such as earliest deadline first (EDF), minimum laxity first (MLF), or maximum urgency first (MUF).

Deterministic real-time systems have traditionally been more amenable to well-understood scheduling analysis techniques. Consequently, our research efforts were initially directed toward static scheduling of deterministic real-time systems. However, the architectural features and optimizations that we studied and developed are applicable to real-time systems with statistical QoS requirements, such as constrained latency multimedia systems or telecom call processing. This section describes the static scheduling service [23] that we developed to support scheduling for hard real-time systems with deterministic QoS requirements.

4.2 Responsibilities of a Real-time Scheduling Service

This subsection examines the analysis capabilities and scheduling policies provided by TAO's Real-time Scheduling Service. This service is responsible for allocating CPU resources to meet the QoS needs of the applications that share the ORB endsystem. For real-time applications with deterministic QoS requirements, the Scheduling Service guarantees that all processing requirements will be met. For real-time applications with statistical QoS requirements, the Scheduling Service tries to meet system processing requirements within the desired tolerance, while also trying to maximize CPU utilization.

The initial design and implementation of TAO's real-time Scheduling Service [23] targeted deterministic real-time applications that require off-line, static scheduling on a single CPU. However, the Scheduling Service is also useful for dynamic and distributed real-time scheduling, as well [46]. Therefore, the Scheduling Service is defined as a CORBA object, *i.e.*, as an implementation of an IDL interface. This design enables the Scheduling Service to be accessed either locally or remotely without having to reimplement clients that use it.

TAO's Real-time Scheduling Service has the following off-line and on-line responsibilities:

Off-line scheduling feasibility analysis: TAO's Scheduling Service performs off-line feasibility analysis of all IDL operations that register with it. This analysis results in a determination of whether there are sufficient CPU resources to perform all requested operations, as discussed in Section 4.5.

Request priority assignment: *Request priority* is the relative priority of a request[5] to any other. It is used by TAO to dispatch requests in order of their priority. *Thread priority* is the priority that corresponds to that of the thread that will invoke the request. During off-line analysis, the Scheduling Service 1) assigns a request priority to each request and 2) assigns each request to one of the preconfigured thread priorities. At run-time, the Scheduling Service provides an interface that allows TAO's real-time ORB endsystem to access these priorities. Priorities are the mechanism for interfacing with the local endsystem's OS dispatcher, as discussed in Section 4.4.

A high-level depiction of the steps involved in the off-line and on-line roles of TAO's Scheduling Service is shown in Figure 10. In step 1, the Scheduling Service constructs graphs of dependent operations using the QoS information registered with it by the application. This QoS information is stored in `RT_Info` structures described in Section 4.3.3. In step 2, it identifies threads by looking at the terminal nodes of these dependency graphs and populates an `RT_Info` repository in step 3. In step 4 it assesses schedulability and assigns priorities, generating the priority tables as compilable C++ code in step 5. These five steps occur off-line during the (static) schedule configuration process. Finally, the priority tables generated in step 5 are used at run-time in step 6 by TAO's ORB endsystem.

TAO's real-time Scheduling Service guarantees that all `RT_Operations` in the system are dispatched with sufficient time to meet their deadlines. To accomplish this, the

[5] A *request* is the run-time representation of an operation in an IDL interface that is passed between client and server.

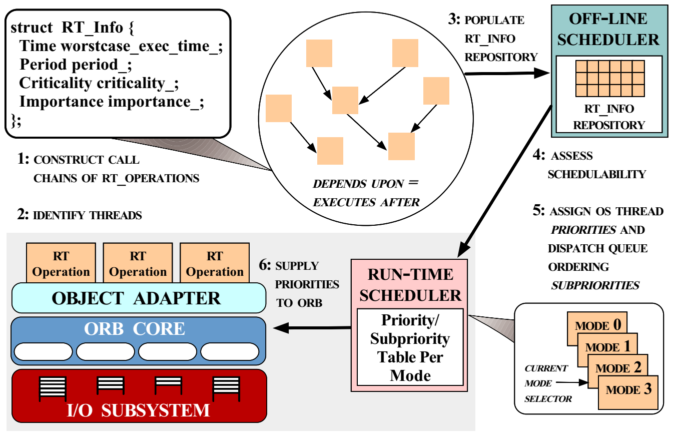
struct RT_Info {
Time worstcase_exec_time_;
Period period_;
Criticality criticality_;
Importance importance_;
};
1: CONSTRUCT CALL CHAINS OF RT_OPERATIONS
2: IDENTIFY THREADS
DEPENDS UPON = EXECUTES AFTER
3: POPULATE RT_INFO REPOSITORY
OFF-LINE SCHEDULER
RT_INFO REPOSITORY
4: ASSESS SCHEDULABILITY
5: ASSIGN OS THREAD PRIORITIES AND DISPATCH QUEUE ORDERING SUBPRIORITIES
RT Operation
RT Operation
RT Operation
OBJECT ADAPTER
ORB CORE
I/O SUBSYSTEM
6: SUPPLY PRIORITIES TO ORB
RUN-TIME SCHEDULER
Priority/ Subpriority Table Per Mode
MODE 0
MODE 1
MODE 2
MODE 3
CURRENT MODE SELECTOR

Figure 10: Steps Involved with Off-line and On-line Scheduling

Scheduling Service can be implemented to perform various real-time scheduling policies. [23] describes the rate monotonic scheduling implementation used by TAO's Scheduling Service.

Below, we outline the information that the service requires to build and execute a feasible system-wide schedule. A feasible schedule is one that is schedulable on the available system resources; in other words, it can be verified that none of the operations in the critical set will miss their deadlines. The *critical set* of operations is the subset of all system operations whose failure to execute before the respective deadline would compromise system integrity.

To simplify the presentation, we focus on ORB scheduling for a single CPU. The distributed scheduling problem is not addressed in this presentation. [46] outlines the approaches we are investigating with TAO.

4.3 Specifying QoS Requirements in TAO using Real-time IDL Schemas

Invoking operations on objects is the primary collaboration mechanism between components in an OO system [15]. However, QoS research at the network and OS layers has not addressed key requirements and usage characteristics of OO middleware. For instance, research on QoS for ATM networks has focused largely on policies for allocating bandwidth on a per-connection basis [29]. Likewise, research on real-time operating systems has focused largely on avoiding priority inversion and non-determinism in synchronization and scheduling mechanisms for multi-threaded applications [13].

Determining how to map the insights and mechanisms produced by QoS work at the network and OS layers onto an OO programming model is a key challenge when adding QoS support to ORB middleware [15, 40]. This subsection describes the real-time OO programming model used by TAO. TAO supports the specification of QoS requirements on a per-operation basis using TAO's real-time IDL schemas.

4.3.1 Overview of QoS Specification in TAO

Several ORB endsystem resources are involved in satisfying application QoS requirements, including CPU cycles, memory, network connections, and storage devices. To support end-to-end scheduling and performance guarantees, real-time ORBs must allow applications to specify their QoS requirements so that an ORB subsystem can guarantee resource availability. In non-distributed, deterministic real-time systems, CPU capacity is typically the scarcest resource. Therefore, the amount of computing time required to process client requests must be determined *a priori* so that CPU capacity can be allocated accordingly. To accomplish this, applications must specify their CPU capacity requirements to TAO's off-line Scheduling Service.

In general, scheduling research on real-time systems that consider resources other than CPU capacity relies upon on-line scheduling [60]. Therefore, we focus on the specification of CPU resource requirements. TAO's QoS mechanism for expressing CPU resource requirements can be readily extended to other shared resources, such as network and bus bandwidth, once scheduling and analysis capabilities have matured.

The remainder of this subsection explains how TAO supports QoS specification for the purpose of CPU scheduling for IDL operations that implement real-time operations. We outline our Real-time IDL (RIDL) schemas: `RT_Operation` interface and its `RT_Info struct`. These schemas convey QoS information, *e.g.*, CPU requirements, to the ORB on a per-operation basis. We believe that this is an intuitive QoS specification model for developers since it maps directly onto the OO programming paradigm.

4.3.2 The RT_Operation Interface

The `RT_Operation` interface is the mechanism for conveying CPU requirements from processing tasks performed by application operations to TAO's Scheduling Service, as shown in the following CORBA IDL interface:[6]

```
module RT_Scheduler
{
  // Module TimeBase defines the OMG Time Service.
  typedef TimeBase::TimeT Time; // 100 nanoseconds
  typedef Time Quantum;

  typedef long Period; // 100 nanoseconds
```

[6]The remainder of the `RT_Scheduler` module IDL description is shown in Section 4.5.1.

```
enum Importance
// Defines the importance of the operation,
// which can be used by the Scheduler as a
// "tie-breaker" when other scheduling
// parameters are equal.
{
  VERY_LOW_IMPORTANCE,
  LOW_IMPORTANCE,
  MEDIUM_IMPORTANCE,
  HIGH_IMPORTANCE,
  VERY_HIGH_IMPORTANCE
};

typedef long handle_t;
// RT_Info's are assigned per-application
// unique identifiers.

struct Dependency_Info
{
  long number_of_calls;
  handle_t rt_info;
  // Notice the reference to the RT_Info we
  // depend on.
};

typedef sequence<Dependency_Info> Dependency_Set;

typedef long OS_Priority;
typedef long Sub_Priority;
typedef long Preemption_Priority;

struct RT_Info
  // = TITLE
  //   Describes the QoS for an "RT_Operation".
  //
  // = DESCRIPTION
  // The CPU requirements and QoS for each
  // "entity" implementing an application
  // operation is described by the following
  // information.
{
  // Application-defined string that uniquely
  // identifies the operation.
  string entry_point_;

  // The scheduler-defined unique identifier.
  handle_t handle_;

  // Execution times.
  Time worstcase_execution_time_;
  Time typical_execution_time_;

  // To account for server data caching.
  Time cached_execution_time_;

  // For rate-base operations, this expresses
  // the rate.  0 means "completely passive",
  // i.e., this operation only executes when
  // called.
  Period period_;

  // Operation importance, used to "break ties".
  Importance importance_;

  // For time-slicing (for BACKGROUND
  // operations only).
    Quantum quantum_;

    // The number of internal threads contained
    // by the operation.
    long threads_;

    // The following attributes are defined by
    // the Scheduler once the off-line schedule
    // is computed.

    // The operations we depend upon.
    Dependency_Set dependencies_;

    // The OS por processing the events generated
    // from this RT_Info.
    OS_Priority priority_;

    // For ordering RT_Info's with equal priority.
    Sub_Priority subpriority_;

    // The queue number for this RT_Info.
    Preemption_Priority preemption_priority_;
  };
};
```

As shown above, the `RT_Operation` interface contains type definitions and its key feature, the `RT_Info struct`, which is described below.

4.3.3 The RT_Info Struct

Applications that use TAO must specify all their scheduled resource requirements. This QoS information is currently provided to TAO before program execution. In the case of CPU scheduling, the QoS requirements are expressed using the following attributes of an `RT_Info` IDL `struct`:

Worst-case execution time: The worst-case execution time, C, is the maximum execution time that the `RT_Operation` requires. It is used in conservative scheduling analysis for applications with strict real-time requirements.

Typical execution time: The typical execution time is the execution time that the `RT_Operation` usually requires. The typical execution time may be useful with some scheduling policies, *e.g.*, statistical real-time systems that can relax the conservative worst-case execution time assumption. However, it is not currently used in TAO's deterministic real-time Scheduling Service.

Cached execution time: If an operation can provide a cached result in response to service requests, then the cached execution time is set to a non-zero value. During execution, for periodic functions, the worst-case execution cost is only incurred once per period if caching is enabled, *i.e.*, if this field is non-zero. The scheduling analysis incorporates caching by only including one term with the worst-case execution time for the operation, per period, no matter how many times it is called, and by using the cached execution time for all other calls.

Period: The period is the minimum time between successive iterations of the operation. If the operation executes as an active object [50] with multiple threads of control, then at least one of those threads must execute at least that often.

A period of 0 indicates that the operation is totally *reactive*, *i.e.*, it does not specify a period. Reactive operations are always called in response to requests by one or more clients. Although the Run-Time Scheduler in TAO need not treat reactive operations as occurring periodically, it must account for their execution time.

Criticality: The operation criticality is an enumeration value ranging from lowest criticality, *i.e.*, VERY_LOW_CRITICALITY, up to highest criticality, *i.e.*, VERY_HIGH_CRITICALITY. Certain scheduling strategies implemented in the Scheduling Service (notably maximum urgency first [49]) consider criticality as the primary distinction between operations when assigning priority.

Importance: The operation importance is an enumeration value ranging from lowest importance, *i.e.*, VERY_LOW_IMPORTANCE, up to highest importance, *i.e.*, VERY_HIGH_IMPORTANCE. The Scheduling Service uses importance as a "tie-breaker" to order the execution of `RT_Operations` when data dependencies or other factors such as criticality do not impose an ordering.

Quantum: Operations within a given priority may be time-sliced, *i.e.*, preempted at any time by the ORB endsystem dispatcher resumed at a later time. If a time quantum is specified for an operation, then that is the maximum time that it will be allowed to run before preemption, if there are any other runnable operations at that priority. This time-sliced scheduling is intended to provide fair access to the CPU for lowest priority operations. Quantum is not currently used in the Scheduling Service.

Dependency Info: This is an array of handles to other `RT_Info` instances, one for each `RT_Operation` that this one directly depends on. The dependencies are used during scheduling analysis to identify threads in the system: each separate dependency graph indicates a thread. In addition, the number of times that the dependent operation is called is specified, for accurate execution time calculation.

The RIDL schemas outlined above can be used to specify the run-time execution characteristics of object operations to TAO's Scheduling Service. This information is used by TAO to (1) validate the feasibility of a schedule and (2) allocate ORB endsystem and network resources to process `RT_Operations`. A single `RT_Info` instance is required for each `RT_Operation`.

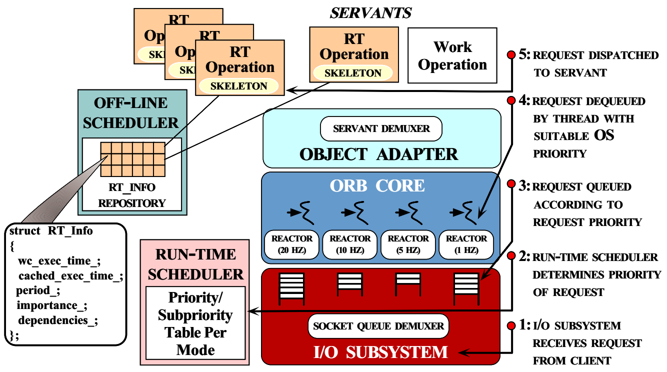
SERVANTS
RT Operation SKELETON
RT Operation SKELETON
Work Operation
OFF-LINE SCHEDULER
RT_INFO REPOSITORY
struct RT_Info { wc_exec_time_; cached_exec_time_; period_; importance_; dependencies_; };
RUN-TIME SCHEDULER
Priority/ Subpriority Table Per Mode
SERVANT DEMUXER
OBJECT ADAPTER
ORB CORE
REACTOR (20 HZ)
REACTOR (10 HZ)
REACTOR (5 HZ)
REACTOR (1 HZ)
SOCKET QUEUE DEMUXER
I/O SUBSYSTEM
5: REQUEST DISPATCHED TO SERVANT
4: REQUEST DEQUEUED BY THREAD WITH SUITABLE OS PRIORITY
3: REQUEST QUEUED ACCORDING TO REQUEST PRIORITY
2: RUN-TIME SCHEDULER DETERMINES PRIORITY OF REQUEST
1: I/O SUBSYSTEM RECEIVES REQUEST FROM CLIENT

Figure 11: TAO Run-time Scheduling Participants

4.4 Overview of TAO's Scheduling Model

TAO's on-line scheduling model includes the following participants, as shown in Figure 11:

Work_Operation: A `Work_Operation` is a unit of work that encapsulates application-level processing or communication activity. For example, utility functions that read input, print output, or convert physical units can be `Work_Operations`. In some real-time environments, a `Work_Operation` is called a *module* or *process*, but we avoid these terms because of their overloaded usage in OO and OS contexts.

RT_Operation: An `RT_Operation` is a type of `Work_Operation` that has timing constraints. Each `RT_Operation` is considered to be an operation defined on a CORBA IDL interface, that has its own QoS information specified in terms of the attributes in its run-time information (`RT_Info`) descriptor. Thus, an application-level object with multiple operations may require multiple `RT_Operation` instances, one for each distinct class of QoS specifications.

Thread: Threads are units of concurrent execution. A thread can be implemented with various threading APIs, *e.g.*, a Solaris or POSIX thread, an Ada task, a VxWorks task, or a Windows NT thread. All threads are contained within `RT_Operations`. An `RT_Operation` containing one or more threads is an *active object* [51]. In contrast, an `RT_Operation` that contains zero threads is a *passive object*. Passive objects only execute in the context of another `RT_Operation`, *i.e.*, they "borrow" the calling operation's thread of control to run.

OS dispatcher: The OS dispatcher uses request priorities to select the next runnable thread that it will assign to a CPU. It removes a thread from a CPU when the thread blocks, and therefore is no longer runnable, or when the thread is *preempted* by a higher priority thread. With *preemptive dispatch-*

ing, any runnable thread with a priority higher than any running thread will preempt a lower priority thread. Then, the higher priority, runnable thread can be dispatched onto the available CPU.

Our analysis assumes *fixed priority*, *i.e.*, the OS does not unilaterally change the priority of a thread. TAO currently runs on a variety of platforms, including real-time operating systems, such as VxWorks and LynxOS, as well as general-purpose operating systems with real-time extensions, such as Solaris 2.x [14] and Windows NT. All these platforms provide fixed priority real-time scheduling. Thus, from the point of view of an OS dispatcher, the priority of each thread is constant. The fixed priority contrasts with the operation of time-shared OS schedulers, which typically *age* long-running processes by decreasing their priority over time [61].

RT_Info: As described in Section 4.3, an `RT_Info` structure specifies an `RT_Operation`'s scheduling characteristics such as computation time and execution period.

Run-Time Scheduler: At run-time, the primary visible vestige of the Scheduling Service is the Run-Time Scheduler. The Run-Time Scheduler maps client requests for particular servant operations into priorities that are understood by the local OS dispatcher. Currently, these priorities are assigned statically prior to run-time and are accessed by TAO's ORB endsystem via an $O(1)$ time table lookup.

4.5 Overview of TAO's Off-line Scheduling Service

To meet the demands of statically scheduled, deterministic real-time systems, TAO's Scheduling Service uses *off-line scheduling*, which has the following two high-level goals:

1. Schedulability analysis: If the operations cannot be scheduled because one or more deadlines could be missed, then the off-line Scheduling Service reports that prior to run-time.

2. Request priority assignment: If the operations can be scheduled, the Scheduling Service assigns a priority to each request. This is the mechanism that the Scheduling Service uses to convey execution order requirements and constraints to TAO's ORB endsystem dispatcher.

4.5.1 Off-line Scheduling Service Interface

The key types and operations of the IDL interface for TAO's off-line Scheduling Service are defined below[7]:

[7]The remainder of the `RT_Scheduler` module IDL description is shown in Section 4.3.2.

```
module RT_Scheduler
{
  exception DUPLICATE_NAME {};
  // The application is trying to
  // register the same task again.

  exception UNKNOWN_TASK {};
  // The RT_Info handle was not valid.

  exception NOT_SCHEDULED {};
  // The application is trying to obtain
  // scheduling information, but none
  // is available.

  exception UTILIZATION_BOUND_EXCEEDED {};
  exception
    INSUFFICIENT_PRIORITY_LEVELS {};
  exception TASK_COUNT_MISMATCH {};
  // Problems while computing off-line
  // scheduling.

  typedef sequence<RT_Info> RT_Info_Set;

  interface Scheduler
    // = DESCRIPTION
    //   This class holds all the RT_Info's
    //   for a single application.
  {
    handle_t create (in string entry_point)
      raises (DUPLICATE_NAME);
    // Creates a new RT_Info entry for the
    // function identifier "entry_point",
    // it can be any string, but the fully
    // qualified name function name is suggested.
    // Returns a handle to the RT_Info.

    handle_t lookup (in string entry_point);
    // Lookups a handle for entry_point.

    RT_Info get (in handle_t handle)
      raises (UNKNOWN_TASK);
    // Retrieve information about an RT_Info.

    void set (in handle_t handle,
              in Time time,
              in Time typical_time,
              in Time cached_time,
              in Period period,
              in Importance importance,
              in Quantum quantum,
              in long threads)
      raises (UNKNOWN_TASK);
    // Set the attributes of an RT_Info.
    // Notice that some values may not
    // be modified (like priority).

    void add_dependency
           (in handle_t handle,
            in handle_t dependency,
            in long number_of_calls)
      raises (UNKNOWN_TASK);
    // Adds <dependency> to <handle>

    void priority
           (in handle_t handle,
            out OS_Priority priority,
          out Sub_Priority subpriority,
          out Preemption_Priority p_priority)
    raises (UNKNOWN_TASK, NOT_SCHEDULED);
  void entry_point_priority
         (in string entry_point,
          out OS_Priority priority,
          out Sub_Priority subpriority,
          out Preemption_Priority p_priority)
    raises (UNKNOWN_TASK, NOT_SCHEDULED);
  // Obtain the run time priorities.

  void compute_scheduling
         (in long minimum_priority,
          in long maximum_priority,
          out RT_Info_Set infos)
    raises (UTILIZATION_BOUND_EXCEEDED,
            INSUFFICIENT_PRIORITY_LEVELS,
            TASK_COUNT_MISMATCH);
  // Computes the scheduling priorities,
  // returns the RT_Info's with their
  // priorities properly filled.  This info
  // can be cached by a Run_Time_Scheduler
  // service or dumped into a C++ file for
  // compilation and even faster (static)
  // lookup.
 };
};
```

Not shown are accessors to system configuration data that the scheduler contains, such as the number of operations and threads in the system.

In general, the Scheduling Service interface need not be viewed by application programmers; the only interface they need to use is the RT_Info interface, described in Section 4.3.3. This division of the Scheduling Service interface into application and privileged sections is shown in Figure 12.

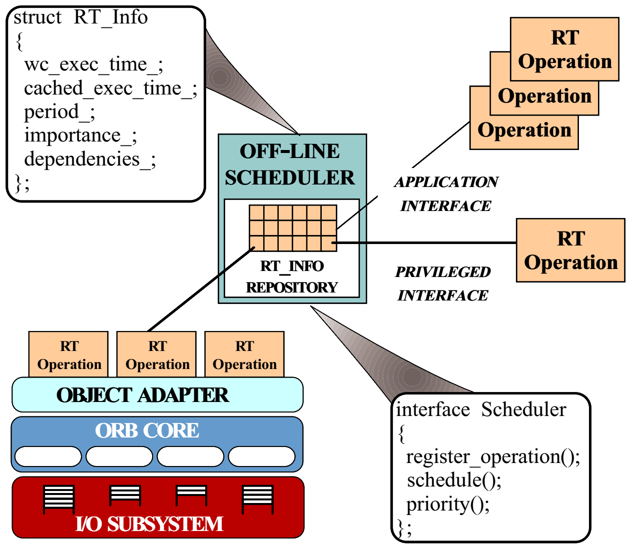

Figure 12: TAO's Two Scheduling Service Interfaces

The privileged interface is only used by common TAO services, such as:

- The Event Channel in TAO's Real-time Event Service [10], which registers its RT_Operations with the off-line Scheduling Service;
- Application-level schedulable operations that do not use the Event Channel;
- TAO's real-time ORB endsystem, which accesses these interfaces to determine client request dispatch priorities.

The remainder of this subsection clarifies the operation of TAO's Scheduling Service, focusing on how it assigns request priorities, when it is invoked, and what is stored in its internal database.

4.5.2 RT_Operation Priority Assignments

The off-line Scheduling Service assigns priorities to each RT_Operation. Because the current implementation of the Scheduling Service utilizes a rate monotonic scheduling policy, priorities are assigned based on an operation's rate. For each RT_Operation in the repository, a priority is assigned based on the following rules:

Rule 1: If the RT_Info::period of an operation is non-zero, TAO's off-line Scheduling Service uses this information to map the period to a thread priority. For instance, 100 msec periods may map to priority 0 (the highest), 200 msec periods may map to priority 1, and so on. With rate monotonic scheduling, for example, higher priorities are assigned to shorter periods.

Rule 2: If the operation does not have a rate requirement, *i.e.*, its RT_Info::period is 0, then its rate requirement must be implied from the operation_dependencies_ field stored in the RT_Info struct. The RT_Info struct with the smallest period, ie, with the fastest rate, in the RT_Info::operation_dependencies_ list will be treated as the operation's implied rate requirement, which is then mapped to a priority. The priority values computed by the off-line Scheduling Service are stored in the RT_Info::priority_ field, which the Run-Time Scheduler can query at run-time via the priority operation.

The final responsibility of TAO's off-line Scheduling Service is to verify the schedulability of a system configuration. This validation process provides a definitive answer to the question "given the current system resources, what is the lowest priority level whose operations all meet their deadlines?" The off-line Scheduling Service uses a repository of RT_Info structures shown in Figure 14 to determine the utilization required by each operation in the system. By comparing the total required utilization for each priority level with the known resources, an assessment of schedulability can be calculated.

TAO's off-line Scheduling Service currently uses the RT_Info attributes of application RT_Operations to build the static schedule and assign priorities according to the following steps:

1. Extract RT_Infos: Extract all `RT_Info` instances for all the `RT_Operations` in the system.

2. Identify real-time threads: Determine all the real-time threads by building and traversing operation dependency graphs.

3. Determine schedulability and priorities: Traverse the dependency graph for each thread to calculate its execution time and periods. Then, assess schedulability based on the thread properties and assign request priorities.

4. Generate request priority table: Generate a table of request priority assignments. This table is subsequently integrated into TAO's run-time system and used to schedule application-level requests.

These steps are described further in the remainder of this section.

4.5.3 Extract RT_Infos

The Scheduling Service is a CORBA object that can be accessed by applications during *configuration runs*. To use the Scheduling Service, users must instantiate one `RT_Info` instantiation for each `RT_Operation` in the system. A configuration run is an execution of the application, TAO, and TAO services which is used to provide the services with any information needed for static configuration. The interactions between the and Scheduling Service during a configuration run are shown in Figure 13.

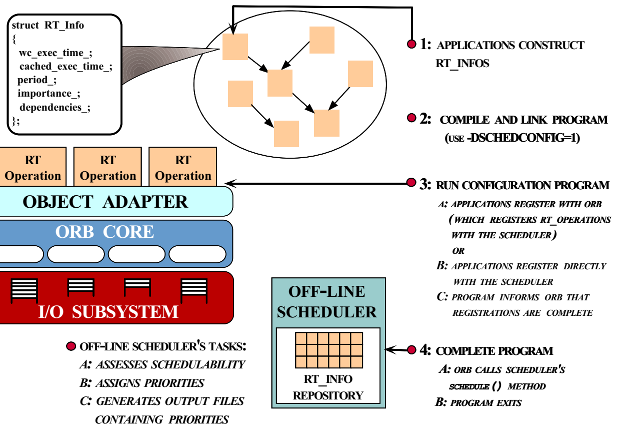
struct RT_Info
{
wc_exec_time_;
cached_exec_time_;
period_;
importance_;
dependencies_;
};
RT Operation
RT Operation
RT Operation
OBJECT ADAPTER
ORB CORE
I/O SUBSYSTEM
OFF-LINE SCHEDULER
RT_INFO REPOSITORY
1: APPLICATIONS CONSTRUCT RT_INFOS
2: COMPILE AND LINK PROGRAM (USE -DSCHEDCONFIG=1)
3: RUN CONFIGURATION PROGRAM
A: APPLICATIONS REGISTER WITH ORB (WHICH REGISTERS RT_OPERATIONS WITH THE SCHEDULER)
OR
B: APPLICATIONS REGISTER DIRECTLY WITH THE SCHEDULER
C: PROGRAM INFORMS ORB THAT REGISTRATIONS ARE COMPLETE
4: COMPLETE PROGRAM
A: ORB CALLS SCHEDULER'S SCHEDULE () METHOD
B: PROGRAM EXITS
OFF-LINE SCHEDULER'S TASKS:
A: ASSESSES SCHEDULABILITY
B: ASSIGNS PRIORITIES
C: GENERATES OUTPUT FILES CONTAINING PRIORITIES

Figure 13: Scheduling Steps During a Configuration Run

The `RT_Info` instantiations, Step 1, are compiled and linked into the main program, Step 2. The application is then executed, Step 3. It registers each `RT_Operation` with either TAO (currently, via TAO's Real-time Event Service), Step 3A, or directly with the Scheduling Service, Step 3B, for operations that do not use TAO. The application notifies TAO, Step 3C, which in turn notifies the Scheduling Service, when all registrations have finished. TAO invokes the off-line scheduling process, Step 4A. Finally, the application exits, Step 4B.

With off-line scheduling, the `RT_Info`s are not needed at run-time. Therefore, one space-saving optimization would be to conditionally compile `RT_Info`s only during configuration runs.

The application should use the `destroy` operation to notify the Scheduling Service when the program is about to exit so that it can release any resources it holds. It is necessary to release memory during configuration runs in order to permit repeated runs on OS platforms, such as VxWorks, that do not release heap-allocated storage when a program terminates.

For consistency in application code, the Scheduling Service configuration and run-time interfaces are identical. The `schedule` operation is essentially a *no-op* in the run-time version; it merely performs a few checks to ensure that all operations are registered and that the number of priority values are reasonable.

4.5.4 Identify Real-time Threads

After collecting all of the `RT_Info` instances, the Scheduling Service identifies threads and performs its schedulability analysis. A *thread* is defined by a directed acyclic graph of `RT_Operations`. An `RT_Info` instance is associated with each `RT_Operation` by the application developer; `RT_Info` creation has been automated using the information available to TAO's Real-time Event Service. `RT_Infos` contain dependency relationships and other information, *e.g.*, *importance*, which determines possible run-time ordering of `RT_Operation` invocations. Thus, a *graph* of dependencies from each `RT_Operation` can be generated mechanically, using the following algorithm:

1. Build a repository of `RT_Info` instances: This task consists of the following two steps:

- Visit each `RT_Info` instance; if not already visited, add to repository, and
- Visit the `RT_Info` of each dependent operation, depth first, and add a link to the dependent operation's internal (to the Scheduling Service) `Dependency_Info` array.

2. Find terminal nodes of dependent operation graphs: As noted in Section 4.5.2, identification of real-time threads involves building and traversing operation dependency graphs. The terminal nodes of separate dependent operation graphs indicate, and are used to identify, threads. The operation dependency graphs capture data dependency, *e.g.*, if operation

A calls operation B, then operation A needs some data that operation B produces, and therefore operation A depends on operation B. If the two operations execute in the context of a single thread, then operation B must execute before operation A. Therefore, the terminal nodes of the dependency graphs delineate threads.

3. Traverse dependent operation graphs: After identifying the terminal nodes of dependent operation graphs, the graphs are traversed to identify the operations that compose each thread. Each traversal starts from a dependent operation graph terminal node, and continues towards the dependent operation's roots until termination. An operation may be part of more than one thread, indicating that each of the threads may call that operation.

The algorithm described above applies several restrictions on the arrangement of operation dependencies. First, a thread may be identified by only one operation; this corresponds directly to a thread having a single entry point. Many OS thread implementations support only a single entry point, *i.e.*, a unique function which is called when the thread is started. This restriction imposes no additional constraints on those platforms.

The second restriction is that cycles are prohibited in dependency relationships. Again, this has a reasonable interpretation. If there was a cycle in a dependency graph, there would be no bound, known to the scheduler, on the number of times the cycle could repeat. To alleviate this restriction, the application can absorb dependency graph cycles into an operation that encapsulates them. Its `RT_Info` would reflect the (bounded) number of internal dependency graph cycles in its worst-case execution time.

The `RT_Info` repository that the Scheduling Service builds is depicted in Figure 14.

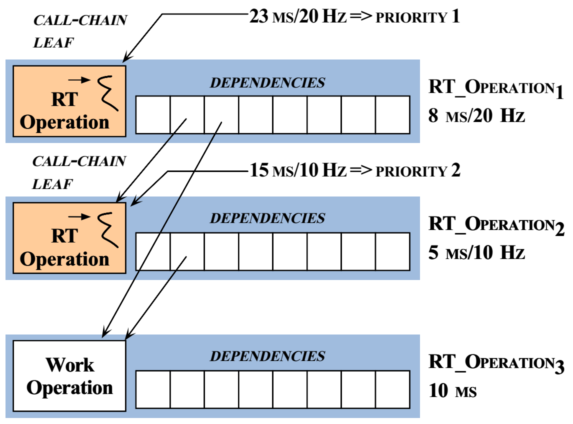

Figure 14: The `RT_Info` Repository

The Scheduling Service's `RT_Info` repository includes the `RT_Info` reference and an array of the `RT_Operations` that it depends upon. These `RT_Operation` dependencies are depicted by blocks with arrows to the dependent operations. The `Dependency_Info` arrays are initialized while first traversing the `RT_Info` instances, to identify threads. Terminal nodes of the dependent operation graphs are identified; these form the starting point for thread identification.

Passive `RT_Operations`, *i.e.*, those without any internal threads of their own, do not appear as terminal nodes of dependent operation graphs. They may appear further down a dependent operation graph, in which case their worst-case and typical execution times are added to the corresponding execution times of the calling thread. However, cached execution times may be added instead, for periodic functions, depending on whether result caching is enabled and whether the operation has been visited already in the current period.

The algorithm for identifying real-time threads may appear to complicate the determination of operation execution times. For instance, instead of specifying a thread's execution time, an operation's execution time must be specified. However, this design is instrumental in supporting an OO programming abstraction that provides QoS specification and enforcement on a per-operation basis. The additional information is valuable to accurately analyze the impact of object-level caching and to provide finer granularity for reusing `RT_Infos`. In addition, this approach makes it convenient to measure the execution times of operations; profiling tools typically provide that information directly.

4.5.5 Determine Schedulability and Priorities

Starting from terminal nodes that identify threads, the `RT_Info` dependency graphs are traversed to determine thread properties, as follows:

Traverse each graph: summing the worst case and typical execution times along the traversal. To determine the period at which the thread must run, save the minimum period of all of the non-zero periods of all of the `RT_Infos` visited during the traversal.

Assign priorities: depending on the scheduling strategy used, higher priority is assigned to higher criticality, higher rate, *etc.*.

Based on the thread properties, and the scheduling strategy used, schedule feasibility is assessed. For example, with RMA, EDF, or MLF, if the total CPU utilization is below the utilization bound, then the schedule for the set of threads is feasible. With MUF, if utilization by all operations in the critical set is below the utilization bound, then the schedule is feasible, even though schedulability of operations outside the critical set may or may not be guaranteed. If the schedule is feasible, request priorities are assigned according to the scheduling strategy, *i.e.*, for RMS requests with higher rates are assigned higher priorities, for MUF requests with higher criticality levels are assigned higher priorities, *etc.*.

4.5.6 Generate Request Priority Table

The Scheduling Service generates a table of request priority assignments. Every thread is assigned a unique integer identifier. This identifier is used at run-time by TAO's ORB endsystem to index into the request priority assignment table. These priorities can be accessed in $O(1)$ time because all scheduling analysis is performed off-line.

Output from the Scheduling Service is produced in the form of an initialized static table that can be compiled and linked into the executable for run-time, *i.e.*, other than configuration, runs. The Scheduling Service provides an interface for the TAO's ORB endsystem to access the request priorities contained in the table.

The initial configuration run may contain, at worst, initial estimates of `RT_Operation` execution times. Likewise, it may include some execution times based on code simulation or manual instruction counts. Successive iterations should include actual measured execution times. The more accurate the input, the more reliable the schedulability assessment.

Off-line configuration runs can be used to fill in the `Dependency_Info` arrays and calibrate the execution times of the `RT_Info` instances for each of the `RT_Operations`. The initial implementation of the Scheduling Service requires that this input be gathered manually. TAO's Real-time Event Service [10] fills in the `Dependency_Info` arrays for its suppliers. Therefore, applications that manage all of their real-time activity through TAO's Event Service do not require manual collection of dependency information.

One user of the Scheduling Service has written a thin layer interface for calibrating the `RT_Info` execution times on VxWorks, which provides a system call for timing the execution of a function. During a configuration run, conditionally compiled code issues that system call for each `RT_Operation` and stores the result in the `RT_Info` structure.

5 Designing a Real-time ORB Core

Section 4 examined the components used by TAO to analyze and generate feasible real-time schedules based on abstract descriptions of CORBA operations. To ensure that these schedules operate correctly at run-time requires an ORB Core that executes operations efficiently and predictably end-to-end. This section describes alternative designs for ORB Core concurrency and connection architectures. Sections 5.1 and 5.2 qualitatively evaluate how the ORB Core connection and concurrency architectures manage the aggregate processing capacity of ORB endsystem components and application operations.

Sections 5.3, 5.4, and 5.5 then present quantitative results that illustrate empirically how the concurrency architectures used by CORBAplus, COOL, MT-Orbix, and TAO perform on Solaris, which is a general-purpose OS with real-time extensions, and Chorus Classix, which is a real-time operating system. CORBAplus and MT-Orbix were not designed to support applications with real-time requirements. The Chorus COOL ORB was designed for embedded systems with small memory footprints. TAO was designed to support real-time applications with deterministic and statistical quality of service requirements, as well as best effort requirements, as described in Section 3.

5.1 Alternative ORB Core Connection Architectures

There are two general strategies for structuring connection architecture in an ORB Core: *multiplexed* and *non-multiplexed*. We describe and evaluate various design alternatives for each approach below, focusing on client-side connection architectures in our examples.

5.1.1 Multiplexed Connection Architectures

Most conventional ORBs multiplex all client requests emanating from a single process through one TCP connection to their corresponding server process. This multiplexed connection architecture is commonly used to build scalable ORBs by minimizing the number of TCP connections open to each server. When multiplexing is used, however, a key challenge is to design an efficient ORB Core connection architecture that supports concurrent `read` and `write` operations.

TCP provides untyped bytestream data transfer semantics. Therefore, multiple threads cannot `read` or `write` from the same socket concurrently. Likewise, `write`s to a socket shared within an ORB process must be serialized. Serialization is typically implemented by having a client thread acquire a lock before writing to a shared socket.

For oneway operations, there is no need for additional locking or processing once a request is sent. Implementing twoway operations over a shared connection is more complicated, however. In this case, the ORB Core must allow multiple threads to concurrently "`read`" from a shared socket endpoint.

If server replies are multiplexed through a single TCP connection then multiple threads cannot `read` simultaneously from that socket endpoint. Instead, the ORB Core must demultiplex incoming replies to the appropriate client thread by using the GIOP sequence number sent with the original client request and returned with the servant's reply.

Several common ways of implementing connection multiplexing to allow concurrent `read` and `write` operations are described below.

Active connection architecture: One approach is the *active connection* architecture shown in Figure 15. An application

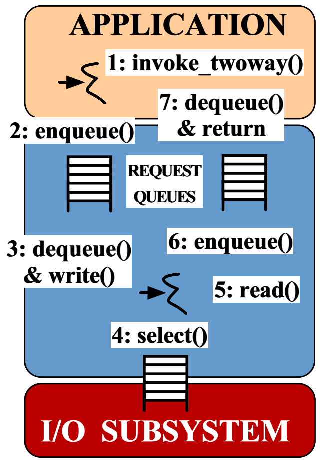

Figure 15: Active Connection Architecture

thread (**1**) invokes a twoway operation, which enqueues the request in the ORB (**2**). A separate thread in the ORB Core services this queue (**3**) and performs a `write` operation on the multiplexed socket. The ORB thread `selects`[8] (**4**) on the socket waiting for the server to reply, `reads` the reply from the socket (**5**), and enqueues the reply in a message queue (**6**). Finally, the application thread retrieves the reply from this queue (**7**) and returns back to its caller.

The advantage of the active connection architecture is that it simplifies ORB implementations by using a uniform queueing mechanism. In addition, if every socket handles packets of the same priority level, *i.e.*, packets of different priorities are not received on the same socket, the active connection can handle these packets in FIFO order without causing request-level priority inversion [17].

The disadvantage with this architecture, however, is that the active connection forces an extra context switch on all twoway operations. To minimize their overhead, many ORBs use a variant of the active connection architecture described next.

Leader/Followers connection architecture: An alternative to the active connection model is the *leader/followers* architecture shown in Figure 16. As before, an application thread invokes a twoway operation call (**1**). Rather than enqueueing the request in an ORB message queue, however, the request is

[8]The `select` call is typically used since a client may have multiple multiplexed connections to multiple servers.

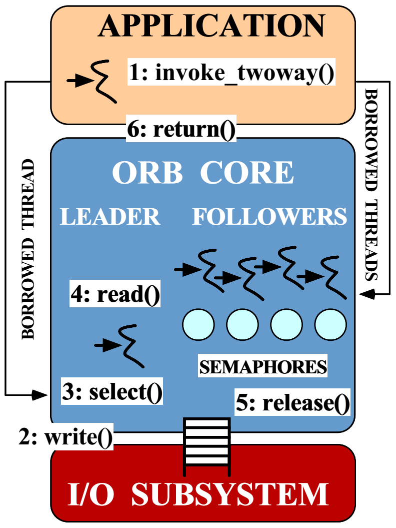

Figure 16: Leader/Follower Connection Architecture

sent across the socket immediately (**2**), using the thread of the application to perform the `write`. Moreover, no single thread in the ORB Core is dedicated to handling all the socket I/O in the leader/follower architecture. Instead, the first thread that attempts to wait for a reply on the multiplexed connection will block in `select` waiting for a reply (**3**). This thread is called the *leader*.

To avoid corrupting the socket bytestream, only the leader thread can `select` on the socket(s). Thus, all client threads that "follow the leader" to `read` replies from the shared socket will block on semaphores managed by the ORB Core. If replies return from the server in FIFO order this strategy is optimal since there is no unnecessary processing or context switching. However, replies may arrive in non-FIFO order. For instance, the next reply arriving from a server could be for any one of the client threads blocked on semaphores.

When the next reply arrives from the server, the leader `reads` the reply (**4**). It uses the sequence number returned in the GIOP reply header to identify the correct thread to receive the reply. If the reply is for the leader's own request, the leader releases the semaphore of the next follower (**5**) and returns to its caller (**6**). The next follower becomes the new leader and blocks on `select`.

If the reply is *not* for the leader, however, the leader must signal the semaphore of the appropriate thread. The signaled thread then wakes up, `reads` its reply, and returns to its caller. Meanwhile, the leader thread continues to `select` for the next reply.

Compared with active connections, the advantage of the leader/follower connection architecture is that it minimizes the number of context switches incurred *if replies arrive in FIFO order*. The drawback, however, is that the complex implementation logic can yield significant locking overhead and priority inversion. The locking overhead stems from the need to acquire mutexes when sending requests and to block on the semaphores while waiting for replies. The priority inversion occurs if the priorities of the waiting threads are not respected by the leader thread when it demultiplexes replies to client threads.

5.1.2 Non-multiplexed Connection Architectures

One technique for minimizing ORB Core priority inversion is to use a non-multiplexed connection architecture, such as the one shown in Figure 17. In this connection architecture, each

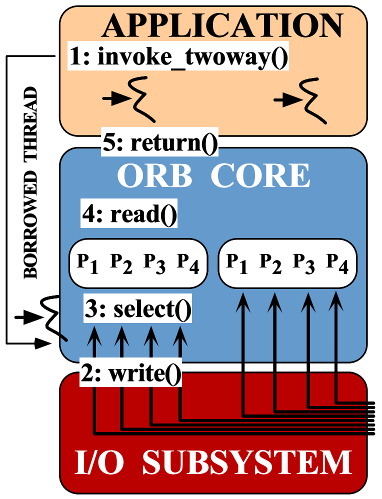

Figure 17: Non-multiplexed Connection Architecture

client thread maintains a table of pre-established connections to servers in thread-specific storage [59]. A separate connection is maintained in each thread for every priority level, *e.g.*, P_1, P_2, P_3, etc. As a result, when a twoway operation is invoked (**1**) it shares no socket endpoints with other threads. Therefore, the `write`, (**2**), `select` (**3**), `read` (**4**), and return (**5**) operations can occur without contending for ORB resources with other threads in the process.

The primary benefit of a non-multiplexed connection architecture is that it preserves end-to-end priorities and minimizes priority inversion while sending requests through ORB endsystems. In addition, since connections are not shared, this design incurs low synchronization overhead because no additional locks are required in the ORB Core when sending/receiving twoway requests.

The drawback with a non-multiplexed connection architecture is that it can use a larger number of socket endpoints than the multiplexed connection model, which may increase the ORB endsystem memory footprint. Therefore, it is most effective when used for statically configured real-time applications, such as avionics mission computing systems [17], which possess a small, fixed number of connections.

5.2 Alternative ORB Core Concurrency Architectures

There are a variety of strategies for structuring the multi-threading architecture in an ORB. Below, we describe a number of alternative ORB Core multi-threading architectures, focusing on server-side multi-threading.

Thread pool is a common architecture for structuring ORB multi-threading, particularly for real-time ORBs [44]. Below, we describe and evaluate several common thread pool architectures.

5.2.1 The Worker Thread Pool Architecture

This ORB multi-threading architecture uses a design similar to the active connection architecture described in Section 5.1.1. As shown in Figure 18, the components in a worker thread

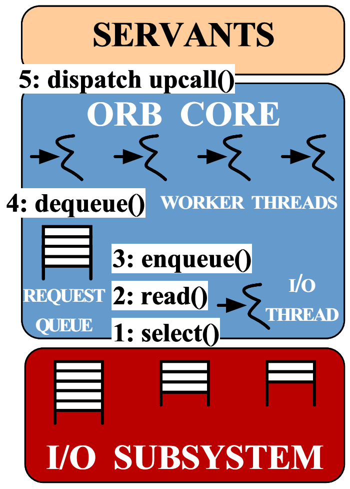

Figure 18: Server-side Worker Thread Pool Multi-threading Architecture

pool include an I/O thread, a request queue, and a pool of worker threads. The I/O thread `selects` (**1**) on the socket endpoints, `reads` (**2**) new client requests, and (**3**) inserts them into the tail of the request queue. A worker thread in the pool dequeues (**4**) the next request from the head of the queue and dispatches it (**5**).

The chief advantage of the worker thread pool multi-threading architecture is its ease of implementation. In particular, the request queue provides a straightforward producer/consumer design. The disadvantages of this model stem from the excessive context switching and synchronization required to manage the request queue, as well as request-level priority inversion caused by connection multiplexing. Since different priority requests share the same transport connection, a high-priority request may wait until a low-priority request that arrived earlier is processed. Moreover, thread-level priority inversions can occur if the priority of the thread that originally `read`s the request is lower than the priority of the servant that processes the request.

5.2.2 The Leader/Follower Thread Pool Architecture

The leader/follower thread pool architecture is an optimization of the worker thread pool model. It is similar to the leader/follower connection architecture discussed in Section 5.1.1. As shown in Figure 19, a pool of threads is allocated

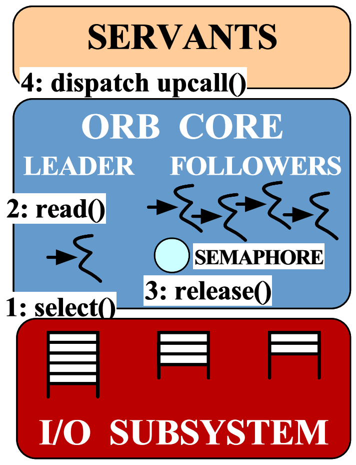

Figure 19: Server-side Leader/Follower Multi-threading Architecture

and a leader thread is chosen to `select` (**1**) on connections for all servants in the server process. When a request arrives, this thread reads (**2**) it into an internal buffer. If this is a valid request for a servant, a follower thread in the pool is released to become the new leader (**3**) and the leader thread dispatches the upcall (**4**). After the upcall is dispatched, the original leader thread becomes a follower and returns to the thread pool. New requests are queued in socket endpoints until a thread in the pool is available to execute the requests.

Compared with the worker thread pool design, the chief advantage of the leader/follower thread pool architecture is that it minimizes context switching overhead incurred by incoming requests. Overhead is minimized since the request need not be transferred from the thread that read it to another thread in the pool that processes it. The disadvantages of the leader/follower architecture are largely the same as with the worker thread design. In addition, it is harder to implement the leader/follower model.

5.2.3 Threading Framework Architecture

A very flexible way to implement an ORB multi-threading architecture is to allow application developers to customize hook methods provided by a *threading framework*. One way of structuring this framework is shown in Figure 20. This de-

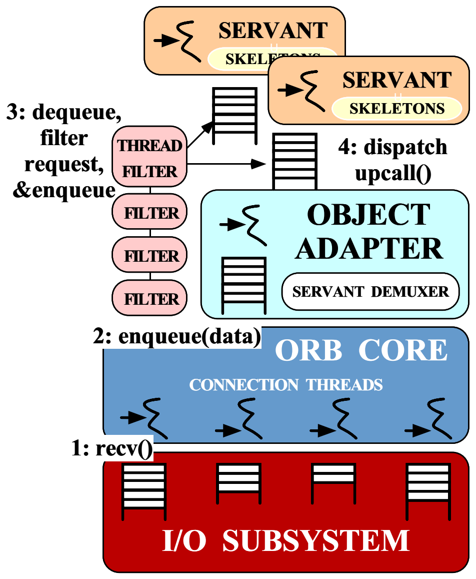

Figure 20: Server-side Thread Framework Multi-threading Architecture

sign is based on the MT-Orbix thread filter framework, which is a variant of the Chain of Responsibility pattern [48].

In MT-Orbix, an application can install a thread filter at the top of a chain of filters. Filters are application-programmable hooks that can perform a number of tasks. Common tasks include intercepting, modifying, or examining each request sent to and from the ORB.

In the thread framework architecture, a connection thread in the ORB Core `reads` (**1**) a request from a socket endpoint and enqueues the request on a request queue in the ORB Core (**2**). Another thread then dequeues the request (**3**) and passes it through each filter in the chain successively. The topmost filter, *i.e.*, the thread filter, determines the thread to handle this request. In the *thread-pool* model, the thread filter enqueues the request into a queue serviced by a thread with the appropriate priority. This thread then passes control back to the ORB, which performs operation demultiplexing and dispatches the upcall (**4**).

The main advantage of a threading framework is its flexibility. The thread filter mechanism can be programmed by server developers to support various multi-threading strategies. For instance, to implement a thread-per-request strategy, the filter can spawn a new thread and pass the request to this new thread. Likewise, the MT-Orbix threading framework can be configured to implement other multi-threading architectures such as thread-per-servant and thread-per-connection.

There are several disadvantages with the thread framework design, however. First, since there is only a single chain of filters, priority inversion can occur because each request must traverse the filter chain in FIFO order. Second, there may be FIFO queueing at multiple levels in the ORB endsystem. Therefore, a high priority request may be processed only after several lower priority requests that arrived earlier. Third, the generality of the threading framework may increase locking overhead, *e.g.*, locks must be acquired to insert requests into the queue of the appropriate thread.

5.2.4 The Reactor-per-Thread-Priority Architecture

The `Reactor`-per-thread-priority architecture is based on the Reactor pattern [43], which integrates transport endpoint demultiplexing and the dispatching of the corresponding event handlers. This threading architecture associates a group of `Reactor`s with a group of threads running at different priorities. As shown in Figure 21, the components in the `Reactor`-per-thread-priority architecture include multiple pre-allocated `Reactor`s, each of which is associated with its own real-time thread of control for each priority level in the ORB. For instance, avionics mission computing systems [10] commonly execute their tasks in fixed priority threads corresponding to the *rates*, *e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz, at which operations are called by clients.

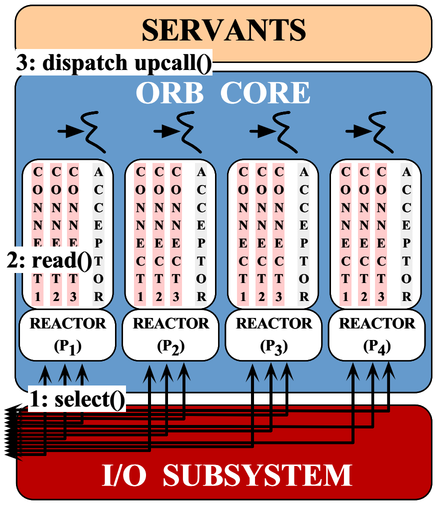

Figure 21: Server-side Reactor-per-Thread-Priority Multi-threading Architecture

Within each thread, the `Reactor` demultiplexes (**1**) all incoming client requests to the appropriate connection handler, *i.e.*, connect$_1$, connect$_2$, etc. The connection handler `reads` (**2**) the request and dispatches (**3**) it to a servant that executes the upcall at its thread priority.

Each `Reactor` in an ORB server thread is also associated with an `Acceptor` [45]. The `Acceptor` is a factory that listens on a particular port number for clients to connect to that thread and creates a connection handler to process the GIOP requests. In the example in Figure 21, there is a listener port for each priority level.

The advantage of the `Reactor`-per-thread-priority architecture is that it minimizes priority inversion and non-determinism. Moreover, it reduces context switching and synchronization overhead by requiring the state of servants to be locked only if they interact across different thread priorities. In addition, this multi-threading architecture supports scheduling and analysis techniques that associate priority with rate, such as Rate Monotonic Scheduling (RMS) and Rate Monotonic Analysis (RMA) [36, 37].

The disadvantage with the `Reactor`-per-thread-priority architecture is that it serializes all client requests for each `Reactor` within a single thread of control, which can reduce parallelism. To alleviate this problem, a variant of this architecture can associate a *pool* of threads with each priority level. Though this will increase potential parallelism, it can incur greater context switching overhead and non-determinism, which may be unacceptable for certain types of real-time applications.

The `Reactor`-per-thread-priority architecture can be integrated seamlessly with the non-multiplexed connection model described in Section 5.1.2 to provide end-to-end priority preservation in real-time ORB endsystems, as shown in Figure 6. In this diagram, the `Acceptor`s listen on ports that correspond to the 20 Hz, 10 Hz, 5 Hz, and 1 Hz rate group thread priorities, respectively. Once a client connects, its `Acceptor` creates a new socket queue and connection handler to service that queue. The I/O subsystem uses the port number contained in arriving requests as a demultiplexing key to associate requests with the appropriate socket queue.

The `Reactor`-per-thread-priority architecture minimizes priority inversion through the entire distributed ORB endsystem by eagerly demultiplexing incoming requests onto the appropriate real-time thread that services the priority level of the target servant. As shown in Section 5.4, this design is well suited for real-time applications with deterministic QoS requirements.

5.3 Benchmarking Testbed

This section describes the experimental testbed we designed to systematically measure sources of latency and throughput overhead, priority inversion, and non-determinism in ORB endsystems. The architecture of our testbed is depicted in Figure 22. The hardware and software components used in the experiments are outlined below.

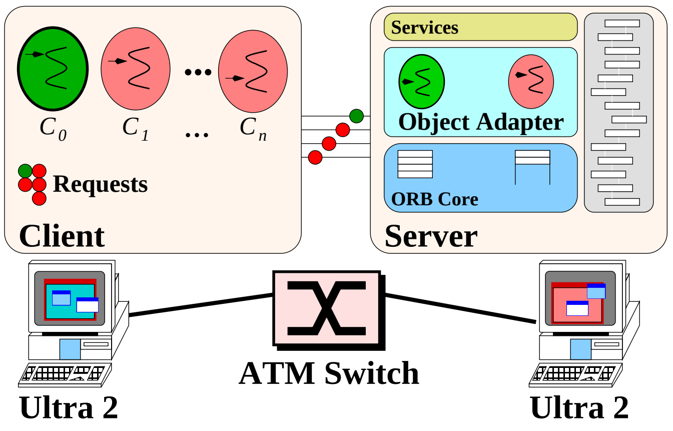

Figure 22: ORB Endsystem Benchmarking Testbed

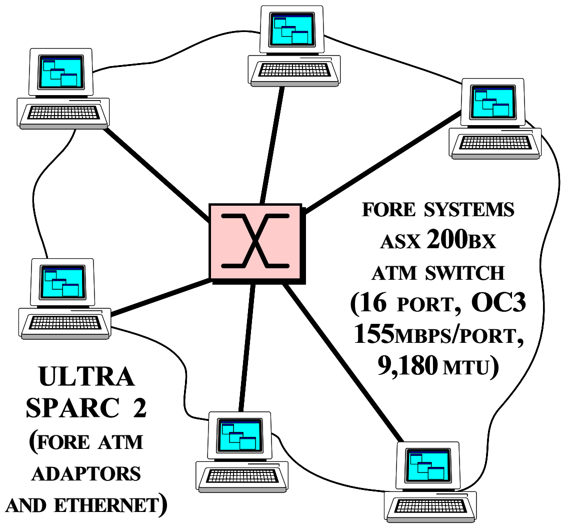
FORE SYSTEMS ASX 200BX ATM SWITCH (16 PORT, OC3 155MBPS/PORT, 9,180 MTU)
ULTRA SPARC 2 (FORE ATM ADAPTORS AND ETHERNET)

Figure 23: Hardware for the CORBA/ATM Testbed

5.3.1 Hardware Configuration

The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running Solaris 2.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSPARC-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The Solaris 2.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework [35].

Each UltraSPARC-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 Kb). This allows up to eight switched virtual connections per card. The CORBA/ATM hardware platform is shown in Figure 23.

5.3.2 Client/Server Configuration and Benchmarking Methodology

Server benchmarking configuration: As shown in Figure 22, our testbed server consists of two servants within an ORB's Object Adapter. One servant runs in a higher priority thread than the other. Each thread processes requests that are sent to its servant by client threads on the other UltraSPARC-2.

Solaris real-time threads [14] are used to implement servant priorities. The high-priority servant thread has the *highest* real-time priority available on Solaris and the low-priority servant has the *lowest* real-time priority.

The server benchmarking configuration is implemented in the various ORBs as follows:

- **CORBAplus:** which uses the worker thread pool architecture described in Section 5.2.1. In version 2.1.1 of CORBAplus, multi-threaded applications have an event dispatcher thread and a pool of worker threads. The dispatcher thread receives the requests and passes them to application worker threads, which process the requests. In the simplest configuration, an application can choose to create no additional threads and rely upon the main thread to process all requests.

- **miniCOOL:** which uses the leader/follower thread pool architecture described in Section 5.2.2. Version 4.3 of miniCOOL allows application-level concurrency control. The application developer can choose between thread-per-request or thread-pool. The thread-pool concurrency architecture was used for our benchmarks since it is better suited than thread-per-request for deterministic real-time applications. In the thread-pool concurrency architecture, the application initially spawns a fixed number of threads. In addition, when the initial thread pool size is insufficient, miniCOOL can be configured to dynamically spawn threads on behalf of server applications to handle requests, up to a maximum limit.

- **MT-Orbix:** which uses the thread pool framework architecture based on the Chain of Responsibility pattern described in Section 5.2.3. Version 2.2 of MT-Orbix is used to create two real-time servant threads at startup. The high-priority thread is associated with the high-priority servant and the low-priority thread is associated with the low-priority servant. Incoming requests are assigned to these threads using the Orbix thread filter mechanism, as shown in Figure 20. Each priority has its own queue of requests to avoid priority inversion within the queue. This inversion could otherwise occur if a high-priority servant and a low-priority servant dequeue requests from the same queue.

• **TAO:** which uses the `Reactor`-per-thread-priority concurrency architecture described in Section 5.2.4. Version 1.0 of TAO integrates the `Reactor`-per-thread-priority concurrency architecture with a non-multiplexed connection architecture, as shown in Figure 21. In contrast, the other three ORBs multiplex all requests from client threads in each process over a single connection to the server process.

Client benchmarking configuration: Figure 22 shows how the benchmarking test used one high-priority client C_0 and n low-priority clients, $C_1 \ldots C_n$. The high-priority client runs in a high-priority real-time OS thread and invokes operations at 20 Hz, *i.e.*, it invokes 20 CORBA twoway calls per second. All low-priority clients have the same lower priority OS thread priority and invoke operations at 10 Hz, *i.e.*, they invoke 10 CORBA twoway calls per second. In each call, the client sends a value of type `CORBA::Octet` to the servant. The servant cubes the number and returns it to the client.

When the test program creates the client threads, they block on a barrier lock so that no client begins work until the others are created and ready to run. When all threads inform the main thread they are ready to begin, the main thread unblocks all client threads. These threads execute in an order determined by the Solaris real-time thread dispatcher. Each client invokes 4,000 CORBA twoway requests at its prescribed rate.

5.4 Performance Results on Solaris

Two categories of tests were used in our benchmarking experiments: *blackbox* and *whitebox*.

Blackbox benchmarks: We computed the average twoway response time incurred by various clients. In addition, we computed twoway operation jitter, which is the standard deviation from the average twoway response time. High levels of latency and jitter are undesirable for real-time applications since they degrade worst-case execution time and reduce CPU utilization. Section 5.4.1 explains the blackbox results.

Whitebox benchmarks: To precisely pinpoint the *sources* of priority inversion and performance non-determinism, we employed whitebox benchmarks. These benchmarks used profiling tools such as UNIX `truss` and `Quantify` [62]. These tools trace and log the activities of the ORBs and measure the time spent on various tasks, as explained in Section 5.4.2.

Together, the blackbox and whitebox benchmarks indicate the end-to-end latency/jitter incurred by CORBA clients and help explain the reason for these results. In general, the results reveal why ORBs like MT-Orbix, CORBAplus, and miniCOOL are not yet suited for applications with real-time performance requirements. Likewise, the results illustrate empirically how and why the non-multiplexed, priority-based ORB Core architecture used by TAO is more suited for many types of real-time applications.

5.4.1 Blackbox Results

As the number of low-priority clients increases, the number of low-priority requests sent to the server also increases. Ideally, a real-time ORB endsystem should exhibit no variance in the latency observed by the high-priority client, irrespective of the number of low-priority clients. Our measurements of end-to-end twoway ORB latency yielded the results in Figure 24.

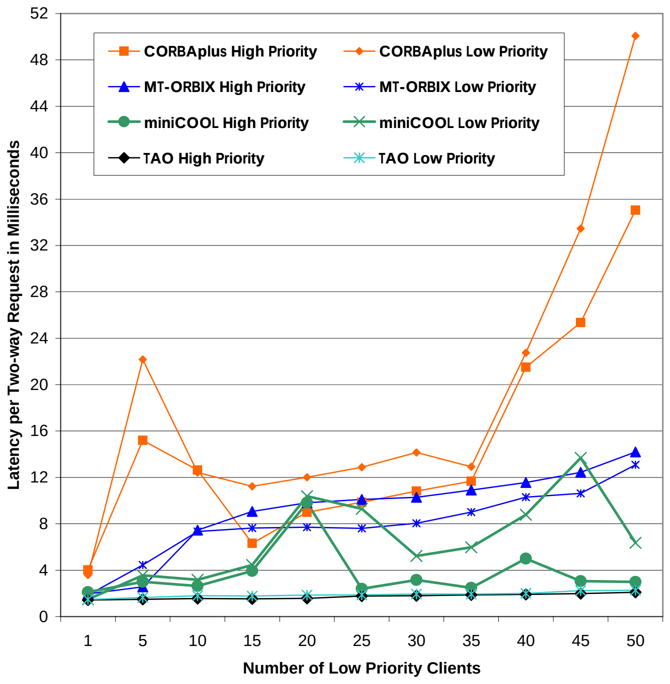

Figure 24: Comparative Latency for CORBAplus, MT-Orbix, miniCOOL, and TAO

Figure 24 shows that as the number of low-priority clients increases, MT-Orbix and CORBAplus incur significantly higher latencies for their high-priority client thread. Compared with TAO, MT-Orbix's latency is 7 times higher and CORBAplus' latency is 25 times higher. Note the irregular behavior of the average latency that miniCOOL displays, *i.e.*, from 10 msec latency running 20 low-priority clients down to 2 msec with 25 low-priority clients. Such non-determinism is clearly undesirable for real-time applications.

The low-priority clients for MT-Orbix, CORBAplus and miniCOOL also exhibit very high levels of jitter. Compared with TAO, CORBAplus incurs 300 times as much jitter and MT-Orbix 25 times as much jitter in the worst case, as shown in Figure 25. Likewise, miniCOOL's low-priority clients display an erratic behavior with several high bursts of jitter, which

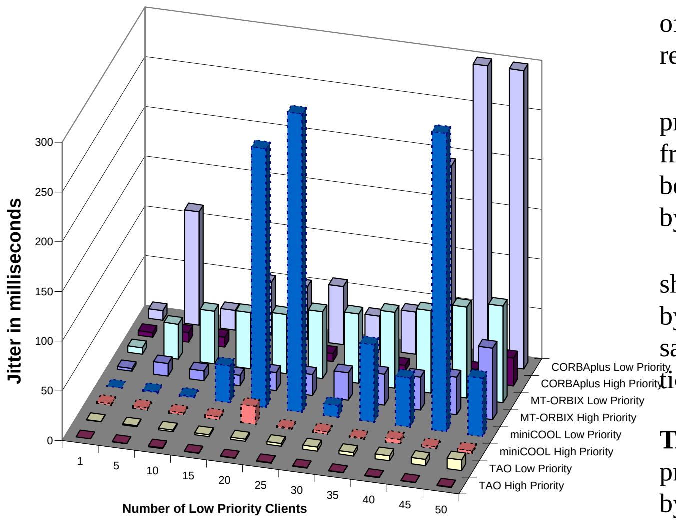

Figure 25: Comparative Jitter for CORBAplus, MT-Orbix, miniCOOL and TAO

makes it undesirable for deterministic real-time applications.

The blackbox results for each ORB are explained below.

CORBAplus results: CORBAplus incurs priority inversion at various points in the graph shown in Figure 24. After displaying a high amount of latency for a small number of low-priority clients, the latency drops suddenly at 10 clients, then eventually rises again. Clearly, this behavior is not suitable for deterministic real-time applications. Section 5.4.2 reveals how the poor performance and priority inversions stem largely from CORBAplus' concurrency architecture. Figure 25 shows that CORBAplus generates high levels of jitter, particularly when tested with 40, 45, and 50 low-priority clients. These results show an erratic and undesirable behavior for applications that require real-time guarantees.

MT-Orbix results: MT-Orbix incurs substantial priority inversion as the number of low-priority clients increase. After the number of clients exceeds 10, the high-priority client performs increasingly worse than the low-priority clients. This behavior is not conducive to deterministic real-time applications. Section 5.4.2 reveals how these inversions stem largely from the MT-Orbix's concurrency architecture on the server. In addition, MT-Orbix produces high levels of jitter, as shown in Figure 25. This behavior is caused by priority inversions in its ORB Core, as explained in Section 5.4.2.

miniCOOL results: As the number of low-priority clients increase, the latency observed by the high-priority client also increases, reaching ∼10 msec, at 20 clients, at which point it decreases suddenly to 2.5 msec with 25 clients. This erratic behavior becomes more evident as more low-priority clients are run. Although the latency of the high-priority client is smaller than the low-priority clients, the non-linear behavior of the clients makes miniCOOL problematic for deterministic real-time applications.

The difference in latency between the high- and the low-priority client is also unpredictable. For instance, it ranges from 0.55 msec to 10 msec. Section 5.4.2 reveals how this behavior stems largely from the connection architecture used by the miniCOOL client and server.

The jitter incurred by miniCOOL is also fairly high, as shown in Figure 25. This jitter is similar to that observed by the CORBAplus ORB since both spend approximately the same percentage of time executing locking operation. Section 5.4.2 evaluates ORB locking behavior.

TAO results: Figure 24 reveals that as the number of low-priority clients increase from 1 to 50, the latency observed by TAO's high-priority client grows by ∼0.7 msecs. However, the difference between the low-priority and high-priority clients starts at 0.05 msec and ends at 0.27 msec. In contrast, in miniCOOL, it evolves from 0.55 msec to 10 msec, and in CORBAplus it evolves from 0.42 msec to 15 msec. Moreover, the rate of increase of latency with TAO is significantly lower than MT-Orbix, Sun miniCOOL, and CORBAplus. In particular, when there are 50 low-priority clients competing for the CPU and network bandwidth, the low-priority client latency observed with MT-Orbix is more than 7 times that of TAO, the miniCOOL latency is ∼3 times that of TAO, and CORBAplus is ∼25 times that of TAO.

In contrast to the other ORBs, TAO's high-priority client always performs better than its low-priority clients. This demonstrates that the connection and concurrency architectures in TAO's ORB Core can maintain real-time request priorities end-to-end. The key difference between TAO and other ORBs is that its GIOP protocol processing is performed on a dedicated connection by a dedicated real-time thread with a suitable end-to-end real-time priority. Thus, TAO shares the minimal amount of ORB endsystem resources, which substantially reduces opportunities for priority inversion and locking overhead.

The TAO ORB produces very low jitter (less than 11 msecs) for the low-priority requests and lower jitter (less than 1 msec) for the high-priority requests. The stability of TAO's latency is clearly desirable for applications that require predictable end-to-end performance.

In general, the blackbox results described above demonstrate that improper choice of ORB Core concurrency and connection software architectures can play a significant role in exacerbating priority inversion and non-determinism. The fact that TAO achieves such low levels of latency and jitter when run over the non-real-time Solaris I/O subsystem further demonstrates the feasibility of using standard OO middleware like CORBA to support real-time applications.

5.4.2 Whitebox Results

For the whitebox tests, we used a configuration of ten concurrent clients similar to the one described in Section 5.3. Nine clients were low-priority and one was high-priority. Each client sent 4,000 twoway requests to the server, which had a low-priority servant and high-priority servant thread.

Our previous experience using CORBA for real-time avionics mission computing [10] indicated that locks constitute a significant source of overhead, non-determinism and potential priority inversion for real-time ORBs. Using `Quantify` and `truss`, we measured the time the ORBs consumed performing tasks like synchronization, I/O, and protocol processing.

In addition, we computed a metric that records the number of calls made to user-level locks (`mutex_lock` and `mutex_unlock`) and kernel-level locks (`_lwp_mutex_lock`, `_lwp_mutex_unlock`, `_lwp_sema_post` and `_lwp_sema_wait`). This metric computes the average number of lock operations per-request. In general, kernel-level locks are considerably more expensive since they incur kernel/user mode switching overhead.

The whitebox results from our experiments are presented below.

CORBAplus whitebox results: Our whitebox analysis of CORBAplus reveals high levels of synchronization overhead from mutex and semaphore operations at the user-level for each twoway request, as shown in Figure 30. Synchronization overhead arises from locking operations that implement the connection and concurrency architecture used by CORBAplus.

As shown in Figure 26, CORBAplus exhibits high synchronization overhead (52%) using kernel-level locks in the client and the server incurs high levels of processing overhead (45%) due to kernel-level lock operations.

For each CORBA request/response, CORBAplus's client ORB performs 199 lock operations, whereas the server performs 216 user-level lock operations, as shown in Figure 30. This locking overhead stems largely from excessive dynamic memory allocation, as described in Section 5.6. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

The CORBAplus connection and concurrency architectures are outlined briefly below.

• **CORBAplus connection architecture:** The CORBAplus ORB connection architecture uses the active connection model described in Section 5.1.1 and depicted in Figure 18. This design multiplexes all requests to the same server through one active connection thread, which simplifies ORB implementations by using a uniform queueing mechanism.

• **CORBAplus concurrency architecture:** The CORBAplus ORB concurrency architecture uses the thread pool architecture described in Section 5.2.1 and depicted in Figure 18. This architecture uses a single I/O thread to `accept` and `read` requests from socket endpoints. This thread inserts the request on a queue that is serviced by a pool of worker threads.

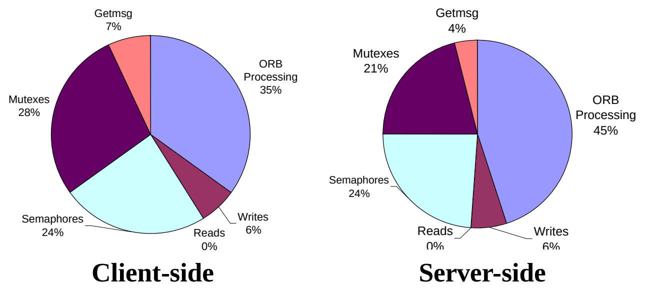

Figure 26: Whitebox Results for CORBAplus

The CORBAplus connection architecture and the server concurrency architecture help reduce the number of simultaneous open connections and simplify the ORB implementation. However, concurrent requests to the shared connection incur high overhead because each send operation incurs a context switch. In addition, on the client-side, threads of different priorities can share the same transport connection, which can cause priority inversion. For instance, a high-priority thread may be blocked until a low-priority thread finishes sending its request. Likewise, the priority of the thread that blocks on the semaphore to receive a reply from a twoway connection may not reflect the priority of the *request* that arrives from the server, thereby causing additional priority inversion.

miniCOOL whitebox results: Our whitebox analysis of miniCOOL reveals that synchronization overhead from mutex and semaphore operations consume a large percentage of the total miniCOOL ORB processing time. As with CORBAplus, synchronization overhead in miniCOOL arises from locking operations that implement its connection and concurrency architecture. Locking overhead accounted for ∼50% on the client-side and more than 40% on the server-side, as shown in Figure 27).

For each CORBA request/response, miniCOOL's client ORB performs 94 lock operations at the user-level, whereas the server performs 231 lock operations, as shown in Figure 30. As with CORBAplus, this locking overhead stems largely from excessive dynamic memory allocation. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

The number of calls per-request to kernel-level locking mechanisms at the server (shown in Figure 31) are unusually

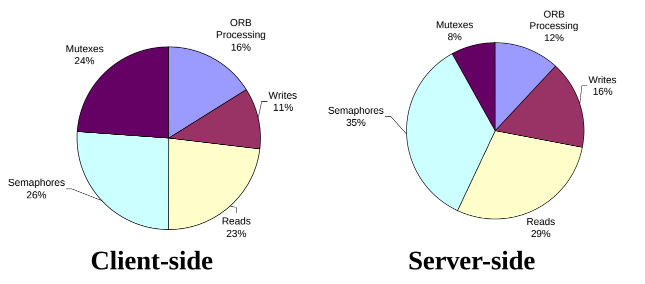

Figure 27: Whitebox Results for miniCOOL

high. This overhead stems from the fact that miniCOOL uses "system scoped" threads on Solaris, which require kernel intervention for all synchronization operations [63].

The miniCOOL connection and concurrency architectures are outlined briefly below.

• **miniCOOL connection architecture:** The miniCOOL ORB connection architecture uses a variant of the leader/followers model described in Section 5.1.1. This architecture allows the leader thread to block in `select` on the shared socket. All following threads block on semaphores waiting for one of two conditions: (1) the leader thread will `read` their reply message and signal their semaphore or (2) the leader thread will `read` its own reply and signal another thread to enter and block in `select`, thereby becoming the new leader.

• **miniCOOL concurrency architecture:** The Sun miniCOOL ORB concurrency architecture uses the leader/followers thread pool architecture described in Section 5.2.2. This architecture waits for connections in a single thread. Whenever a request arrives and validation of the request is complete, the leader thread (1) signals a follower thread in the pool to wait for incoming requests and (2) services the request.

The miniCOOL connection architecture and the server concurrency architecture help reduce the number of simultaneous open connections and the amount of context switching when replies arrive in FIFO order. As with CORBAplus, however, this design yields high levels of priority inversion. For instance, threads of different priorities can share the same transport connection on the client-side. Therefore, a high-priority thread may block until a low-priority thread finishes sending its request. In addition, the priority of the thread that blocks on the semaphore to access a connection may not reflect the priority of the *response* that arrives from the server, which yields additional priority inversion.

MT-Orbix whitebox results: Figure 28 shows the whitebox results for the client-side and server-side of MT-Orbix.

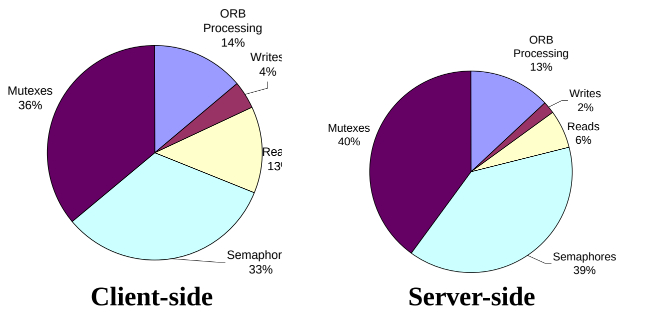

Figure 28: Whitebox Results for MT-Orbix

• **MT-Orbix connection architecture:** Like miniCOOL, MT-Orbix uses the leader/follower multiplexed connection architecture. Although this model minimizes context switching overhead, it causes intensive priority inversions.

• **MT-Orbix concurrency architecture:** In the MT-Orbix implementation of our benchmarking testbed, multiple servant threads were created, each with the appropriate priority, *i.e.*, the high-priority servant had the highest priority thread. A thread filter was then installed to look at each request, determine the priority of the request (by examining the target object), and pass the request to the thread with the correct priority. The thread filter mechanism is implemented by a high-priority real-time thread to minimize dispatch latency.

The thread pool instantiation of the MT-Orbix mechanism described in Section 5.2.3 is flexible and easy to use. However, it suffers from high levels of priority inversion and synchronization overhead. MT-Orbix provides only *one* filter chain. Thus, all incoming requests must be processed sequentially by the filters before they are passed to the servant thread with an appropriate real-time priority. As a result, if a high-priority request arrives after a low-priority request, it must wait until the low-priority request has been dispatched before the ORB processes it.

In addition, a filter can only be called after (1) GIOP processing has completed and (2) the Object Adapter has determined the target object for this request. This processing is serialized since the MT-Orbix ORB Core is unaware of the request priority. Thus, a higher priority request that arrived after a low-priority request must wait until the lower priority request has been processed by MT-Orbix.

MT-Orbix's concurrency architecture is chiefly responsible for its substantial priority inversion shown in Figure 24. This figure shows how the latency observed by the high-priority client increases rapidly, growing from ∼2 msecs to ∼14 msecs as the number of low-priority clients increase from 1 to 50.

The MT-Orbix filter mechanism also causes an increase in synchronization overhead. Because there is just one filter chain, concurrent requests must acquire and release locks to be processed by the filter. The MT-Orbix client-side performs 175 user-level lock operations per-request, while the server-side performs 599 user-level lock operations per-request, as shown in Figure 30. Moreover, MT-Orbix displays a high number of kernel-level locks per-request, as shown in Figure 31.

TAO whitebox results: As shown in Figure 29, TAO exhibits negligible synchronization overhead. TAO performs 40

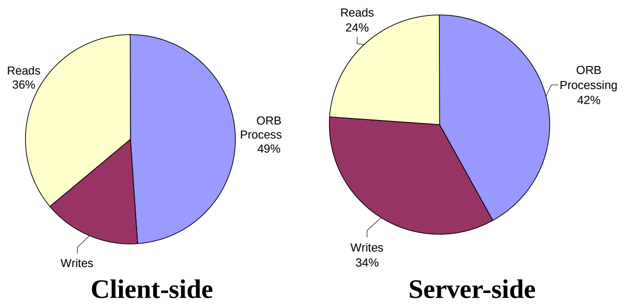

Figure 29: Whitebox Results for TAO

user-level lock operations per-request on the client-side, and 32 user-level lock operations per-request on the server-side. This low amount of synchronization results from the design of TAO's ORB Core, which allocates a separate connection for each priority, as shown in Figure 6. Therefore, TAO's ORB Core minimizes additional user-level locking operations per-request and uses no kernel-level locks in its ORB Core.

• **TAO connection architecture:** TAO uses a non-multiplexed connection architecture, which pre-establishes connections to servants, as described in Section 5.1.2. One connection is pre-established for each priority level, thereby avoiding the non-deterministic delay involved in dynamic connection setup. In addition, different priority levels have their own connection. This design avoids request-level priority inversion, which would otherwise occur from FIFO queueing *across* client threads with different priorities.

• **TAO concurrency architecture:** TAO supports several concurrency architectures, as described in [17]. The `Reactor`-per-thread-priority architecture described in Section 5.2.4 was used for the benchmarks in this paper. In this concurrency architecture, a separate thread is created for each priority level, *i.e.*, each rate group. Thus, the low-priority client issues CORBA requests at a lower rate than the high-priority client (10 Hz vs. 20 Hz, respectively).

On the server-side, client requests sent to the high-priority servant are processed by a high-priority real-time thread. Likewise, client requests sent to the low-priority servant are handled by the low-priority real-time thread. Locking overhead is minimized since these two servant threads share minimal ORB resources, *i.e.*, they have separate `Reactor`s, `Acceptor`s, Object Adapters, etc. In addition, the two threads service separate client connections, thereby eliminating the priority inversion that would otherwise arises from connection multiplexing, as exhibited by the other ORBs we tested.

Locking overhead: Our whitebox tests measured user-level locking overhead (shown in Figure 30) and kernel-level lock-

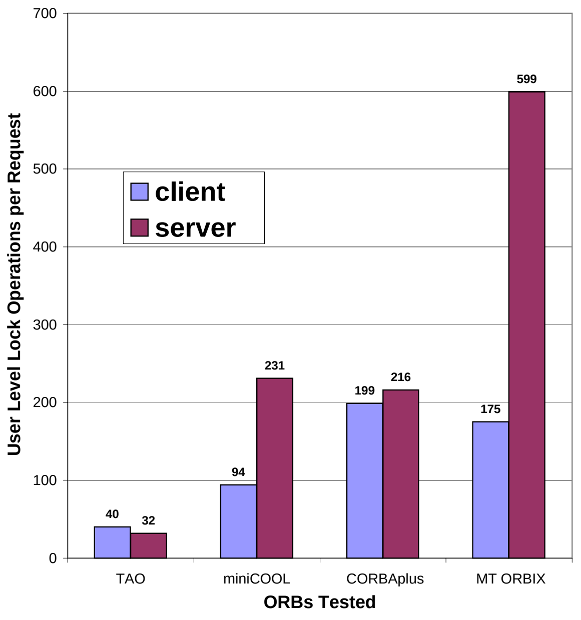

Figure 30: User-level Locking Overhead in ORBs

ing overhead (shown in Figure 31) in the CORBAplus, MT-Orbix, miniCOOL, and TAO ORBs. User-level locks are typically used to protect shared resources within a process. A common example is dynamic memory allocation using global C++ operators `new` and `delete`. These operators allocate memory from a globally managed heap in each process.

Kernel-level locks are more expensive since they typically require mode switches between user-level and the kernel. The semaphore and mutex operations depicted in the whitebox results for the ORBs evaluated above arise from kernel-level lock operations.

TAO limits user-level locking by using buffers that are pre-allocated off the run-time stack. This buffer is subdivided to accommodate the various fields of the request. Kernel-level

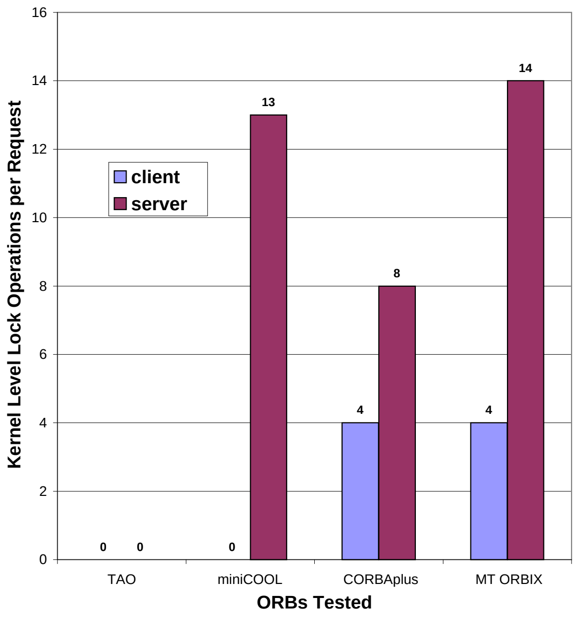

Figure 31: Kernel-level Locking Overhead in ORBs

locking is minimized since TAO can be configured so that ORB resources are not shared between its threads.

5.5 Performance Results on Chorus ClassiX

The performance results in Section 5.4 were obtained on Solaris 2.5.1, which provides real-time scheduling but not real-time I/O [14]. Therefore, Solaris cannot guarantee the availability of resources like I/O buffers and network bandwidth [17]. Moreover, the scheduling performed by the Solaris I/O subsystem is not integrated with the rest of its resource management strategies.

So-called real-time operating systems (RTOS)s typically provide mechanisms for priority-controlled access to OS resources. This allows applications to ensure that QoS requirements are met. RTOS QoS mechanisms typically include real-time scheduling classes that enforce QoS usage policies, as well as real-time I/O to specify processing requirements and operation periods.

Chorus[9] ClassiX is a real-time OS that can scale down to small embedded configurations, as well as scale up to distributed POSIX-compliant platforms [64]. ClassiX provides a real-time scheduler that supports several scheduling algorithms, including priority-based FIFO preemptive scheduling. It supports real-time applications and general-purpose applications.

The IPC mechanism used on ClassiX, Chorus IPC, provides an efficient, location-transparent message-based communication facility on a single board and between multiple interconnected boards. In addition, ClassiX has a TCP/IP protocol stack, accessible via the Socket API, that enables internetworking connectivity with other OS platforms.

To determine the impact of a real-time OS on ORB performance, this subsection presents blackbox results for TAO and miniCOOL using ClassiX.

[9]Chorus has been purchased by Sun Microsystems.

5.5.1 Hardware Configuration:

The following experiments were conducted using two MVME177 VMEbus single-board computers. The MVME177 contains a 60 MHz MC68060 processor and 64 Mbytes of RAM. The MVME177 boards are mounted on a MVME954A 6-slot, 32-bit, VME-compatible backplane. In addition, each MVME177 module has an 82596CA Ethernet transceiver interface.

5.5.2 Software Configuration:

The experiments were run on version 3.1 of ClassiX. The ORBs benchmarked were miniCOOL 4.3 and TAO 1.0. The client/server configurations run were (1) locally, *i.e.*, client and server on one board and (2) remotely, *i.e.*, between two MVME177 boards on the same backplane.

The client/server benchmarking configuration implemented is the same[10] as the one run on Solaris 2.5.1 that is described in Section 5.3.2. MiniCOOL was configured to use the Chorus IPC communication facility to send messages on one board or across boards. This is more efficient than the TCP/IP protocol stack . In addition, we conducted benchmarks of miniCOOL and TAO using the TCP protocol. In general, miniCOOL performs more predictably using Chorus IPC as its transport mechanism.

5.5.3 Blackbox results:

We computed the average twoway response time incurred by various clients. In addition, we computed twoway operation jitter. High levels of latency and jitter are undesirable for real-time applications since they complicate the computation of worst-case execution time and reduce CPU utilization.

miniCOOL using Chorus IPC: As the number of low-priority clients increase, the latency observed by the remote high- and low-priority client also increases. It reaches $\sim$34

[10]Note the number of low-priority clients used was 5 rather than 50 due to a bug in ClassiX that caused `select` to fail if used to wait for events on more than 16 sockets.

msec, increasing linearly, when the client and the server are on different processor boards (remote) as shown in Figure 32.

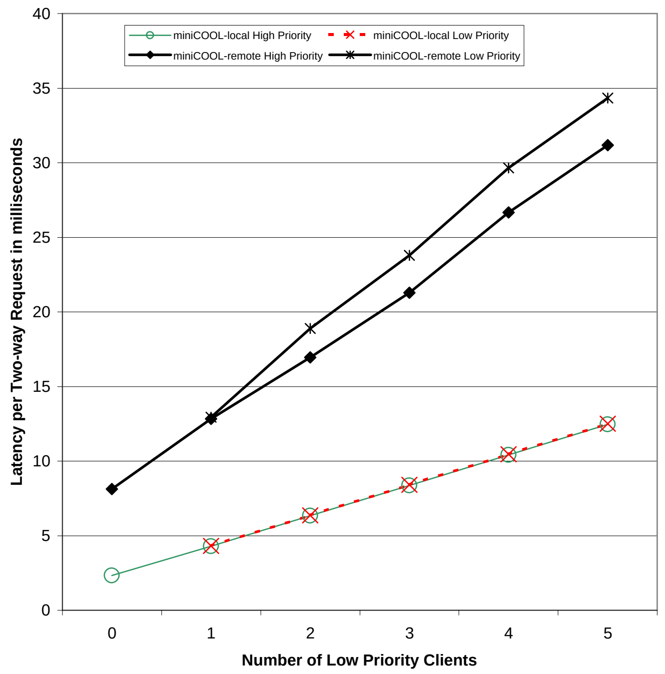

Figure 32: Latency for miniCOOL with Chorus IPC on ClassiX

When the client and server are collocated, the behavior is more stable on both the high and low-priority client, *i.e.*, they are essentially identical since their lines in Figure 32 overlap. The latencies start at ∼2.5 msec of latency and reaches ∼12.5 msecs. Both high- and low-priority clients incur approximately the same average latency.

In all cases, the latency for the high-priority client is always lower than the latency for the low-priority client. Thus, there is no significant priority inversion, which is expected for a real-time system. However, there is still variance in the latency observed by the high-priority client, in both, the remote and local configurations.

In general, miniCOOL performs more predictably on ClassiX than its version for Solaris. This is due to the use of TCP on Solaris versus Chorus IPC on ClassiX. The Solaris latency and jitter results were relatively erratic, as shown in the black-box results from Solaris described in Section 5.4.1.

Figure 33 shows that as the number of low-priority clients increases, the jitter increases progressively manner, for remote high- and low-priority clients. In addition, Figure 33 illustrates that the jitter incurred by miniCOOL's remote clients is fairly high. The unpredictable behavior of high- and low-priority clients is more evident when the client and the server run on separate processor boards, as shown in Figure 32. Moreover,

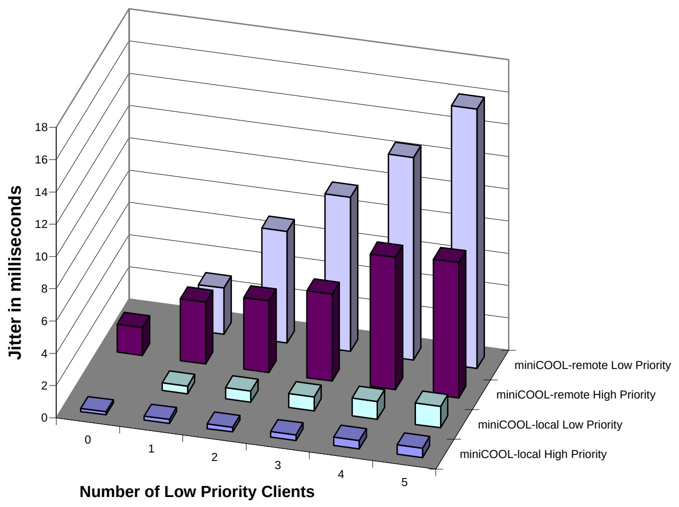

Figure 33: Jitter for miniCOOL with Chorus IPC on ClassiX

Figure 32 illustrates the difference in latency between the local and remote configurations, which appears to stem from the latency incurred by the network I/O driver.

miniCOOL using TCP: We also configured the miniCOOL client/server benchmark to use the Chorus TCP/IP protocol stack. The TCP/IP implementation on ClassiX is not as efficient as Chorus IPC. However, it provided a base for comparison between miniCOOL and TAO (which uses TCP as its transport protocol).

The results we obtained for miniCOOL over TCP show that as the number of low-priority clients increase, the latency observed by the remote high- and low-priority client also increased linearly. The maximum latency was ∼59 msec, when the client and the server are on the same processor board (local) as shown in Figure 34.

The increase in latency for the local configuration is unusual since one would expect the ORB to perform best when client and server are collocated on the same processor. However, when client and server reside in different processor boards, illustrated in Figure 35, the average latency was more stable. This appears to be due to the implementation of the TCP/IP protocol stack, which may not to be optimized for local IPC.

When the client and server are on separate boards, the behavior is similar to the remote clients using Chorus IPC. This indicates that at some of the bottlenecks reside in the Ethernet driver.

In all cases, the latency for the high-priority client is always lower than the latency for the low-priority client, *i.e.*, there appears to be no significant priority inversion, which is expected for a real-time system. However, there is still variance in the latency observed by the high-priority client, in both the remote and local configurations, as shown in Figure 36. The remote configurations incurred the highest vari-

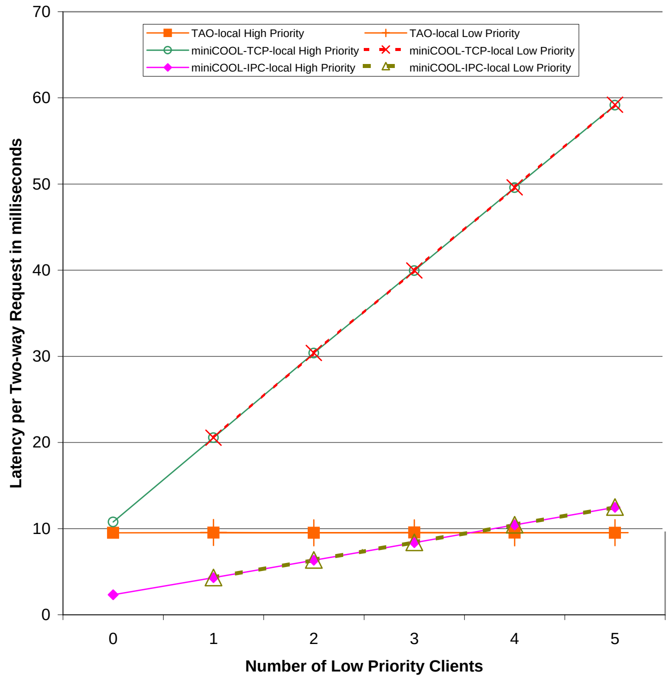

Figure 34: Latency for miniCOOL-TCP, miniCOOL-IPC, and TAO-TCP on ClassiX, local configuration

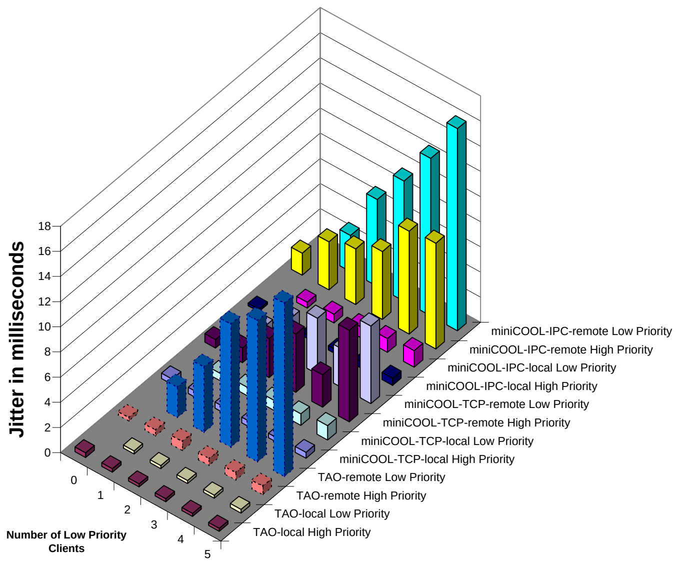

Figure 36: Jitter for miniCOOL-TCP, miniCOOL-IPC and TAO-TCP on ClassiX

ance, with the exception of TAO's remote high-priority clients, whose jitter remained fairly stable. This stability stems from TAO's `Reactor`-per-thread-priority concurrency architecture described in Section 5.2.4.

TAO using TCP: Figure 34 reveals that as the number of low-priority clients increase from 0 to 5, the latency observed by TAO's high-priority client grows by ∼0.005 msecs for the local configuration and Figure 35 shows ∼1.022 msecs for the remote one. Although the remote high-priority client performs as well as the local one, the difference between the low-priority and high-priority remote clients evolves from 0 msec to 6 msec. This increase is unusual and appears to stem from factors external to the ORBn such as the scheduling algorithm and network latency. In general, TAO performs more predictably in other platforms tested with higher bandwidth, *e.g.* 155 Mbps ATM networks. The local client/server test, in contrast, perform very predictably and have little increase in latency.

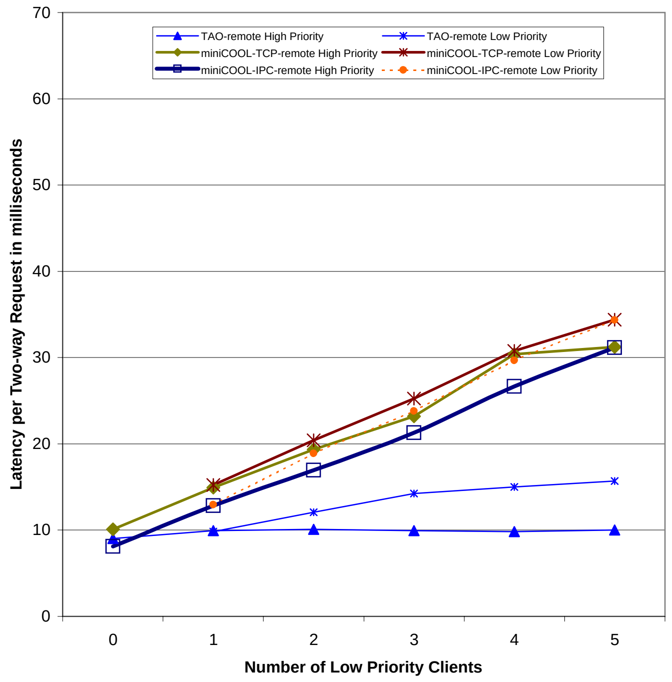

Figure 35: Latency for miniCOOL-TCP, miniCOOL-IPC, and TAO-TCP on ClassiX, remote configuration

The TAO ORB produces very low jitter, less than 2 msecs, for the low-priority requests and lower jitter (less than 1 msec) for the high-priority requests. On this platform, the exception is the remote low-priority client, which may be attributed to the starvation of the low-priority clients by the high-priority one, and the latency incurred by the network. The stability of TAO's latency is clearly desirable for applications that require predictable end-to-end performance.

5.6 Evaluation and Recommendations

The results of our benchmarks illustrate the non-deterministic performance incurred by applications running atop conventional ORBs. In addition, the results show that priority inversion and non-determinism are significant problems in conventional ORBs. As a result, these ORBs are not currently suitable for applications with deterministic real-time requirements. Based on our results, and our prior experience [20, 21, 19, 16] measuring the performance of CORBA ORB endsystems, we suggest the following recommendations to decrease non-determinism and limit priority inversion in real-time ORB endsystems.

1. Real-time ORBs should avoid dynamic connection establishment: ORBs that establish connections dynamically suffer from high jitter. Thus, performance seen by individual clients can vary significantly from the average. Neither CORBAplus, miniCOOL, nor MT-Orbix provide APIs for pre-establishing connections; TAO provides these APIs as extensions to CORBA.

We recommend that APIs to control the pre-establishment of connections should be defined as an OMG standard for real-time CORBA [65, 41].

2. Real-time ORBs should minimize dynamic memory management: Thread-safe implementations of dynamic memory allocators require user-level locking. For instance, the C++ `new` operator allocates memory from a global pool shared by all threads in a process. Likewise, the C++ `delete` operation, which releases allocated memory, also requires user-level locking to update the global shared pool. This lock sharing contributes to the overhead shown in Figure 30. In addition, locking also increases non-determinism due to contention and queueing.

We recommend that real-time ORBs avoid excessive sharing of dynamic memory locks via the use of mechanisms such as thread-specific storage [59], which allocates memory from separate heaps that are unique to each thread.

3. Real-time ORBs should avoid multiplexing requests of different priorities over a shared connection: Sharing connections among multiple threads requires synchronization. Not only does this increase locking overhead, but it also increases opportunities for priority inversion. For instance, high-priority requests can be blocked until low-priority threads release the shared connection lock. Priority inversion can be further exacerbated if multiple threads with multiple levels of thread priorities share common locks. For instance, medium priority threads can preempt a low-priority thread that is holding a lock required by a high-priority thread, which can lead to unbounded priority inversion [13].

We recommend that real-time ORBs allow application developers to determine whether requests with different priorities are multiplexed over shared connections. Currently, neither miniCOOL, CORBAplus, nor MT-Orbix support this level of control, though TAO provides this model by default.

4. Real-time ORB concurrency architectures should be flexible, efficient, and predictable: Many ORBs, such as miniCOOL and CORBAplus, create threads on behalf of server applications. This design is inflexible since it prevents application developers from customizing ORB performance via a different concurrency architecture. Conversely, other ORB concurrency architectures are flexible, but inefficient and unpredictable, as shown by Section 5.4.2's explanation of the MT-Orbix performance results. Thus, a balance is needed between flexibility and efficiency.

We recommend that real-time ORBs provide APIs that allow application developers to select concurrency architectures that are flexible, efficient, *and* predictable. For instance, TAO offers a range of concurrency architectures, such as `Reactor`-per-thread-priority, thread pool, and thread-per-connection. Developers can configure TAO [25] to minimize unnecessary sharing of ORB resources by using thread-specific storage.

5. Real-time ORBs should avoid reimplementing OS mechanisms: Conventional ORBs incur substantial performance overhead because they reimplement native OS mechanisms for endpoint demultiplexing, queueing, and concurrency control. For instance, much of the priority inversion and non-determinism miniCOOL, CORBAplus, and MT-Orbix stem from the complexity of their ORB Core mechanisms for multiplexing multiple client threads through a single connection to a server. These mechanism reimplement the connection management and demultiplexing features in the OS in a manner that (1) increases overhead and (2) does not consider the priority of the threads that make the requests for twoway operations.

We recommend that real-time ORB developers attempt to use the native OS mechanisms as much as possible, *e.g.*, designing the ORB Core to work in concert with the underlying mechanisms rather than reimplementing them at a higher level. A major reason that TAO performs predictably and efficiently is because the connection management and concurrency model used in its ORB Core is closely integrated with the underlying OS features.

6. The design of real-time ORB endsystem architectures should be guided by empirical performance benchmarks: Our prior research on pinpointing performance bottlenecks and optimizing middleware like Web servers [66, 67] and CORBA ORBs [21, 20, 16, 19] demonstrates the efficacy of a measurement-driven research methodology.

We recommend that the OMG adopt standard real-time CORBA benchmarking techniques and metrics. These benchmarks will simplify communication between researchers and developers. In addition, they will facilitate the comparison of performance results and real-time ORB behavior patterns between different ORBs and different OS/hardware platforms. The real-time ORB benchmarking test suite described in this section is available at `www.cs.wustl.edu/~schmidt/TAO.html`.

6 Using Patterns to Build TAO's Extensible ORB Software Architecture

The preceding sections in this paper focused largely on the QoS requirements for real-time ORB endsystems and described how TAO's scheduling, connection, and concurrency architectures are structured to meet these requirements. This section delves deeper into TAO's software architecture by exploring the *patterns* its uses to create *dynamically configurable* real-time ORB middleware.

A pattern represents a recurring solution to a software development problem within a particular context [48, 68]. Patterns help to alleviate the continual re-discovery and re-invention of software concepts and components by capturing solutions to standard software development problems [69]. For instance, patterns are useful for documenting the structure and participants in common communication software micro-architectures like Reactors [43], Active Objects [51], and Brokers [68]. These patterns are generalizations of object-structures that have proven useful to build flexible and efficient event-driven and concurrent communication software such as ORBs.

To focus the discussion, this section illustrates how we have applied patterns to develop TAO. A novel aspect of TAO is its extensible ORB design, which can be customized dynamically to meet specific application QoS requirements and network/endsystem characteristics. As a result, TAO can be extended and maintained more easily than conventional *statically configured* ORBs.

6.1 Why We Need Dynamically Configurable Middleware

A key motivation for ORB middleware is to offload complex distributed system infrastructure tasks from application developers to ORB developers. ORB developers are responsible for implementing reusable middleware components that handle common tasks, such as interprocess communication, concurrency, transport endpoint demultiplexing, scheduling, and dispatching. These components are typically compiled into a run-time ORB library, linked with application objects that use the ORB components, and executed in one or more OS processes.

Although this separation of concerns can simplify application development, it can also yield inflexible and inefficient applications and middleware architectures. The primary reason is that many conventional ORBs are configured *statically* at compile-time and link-time by ORB developers, rather than *dynamically* at installation-time or run-time by application developers. Statically configured ORBs have the following drawbacks [70, 50]:

Inflexibility: Statically-configured ORBs tightly couple each component's *implementation* with the *configuration* of internal ORB components, *i.e.*, which components work together and how they work together. As a result, extending statically-configured ORBs requires modifications to existing source code, which may not be accessible to application developers.

Even if source code is available, extending statically-configured ORBs requires recompilation and relinking. Moreover, any currently executing ORBs and their associated objects must be shutdown and restarted. This static reconfiguration process is not well-suited for application domains like telecom call processing that require 7×24 availability.

Inefficiency: Statically-configured ORBs can be inefficient, both in terms of space and time. Space inefficiency can occur if unnecessary components are always statically configured into an ORB. This can increase the ORB's memory footprint, forcing applications to pay a space penalty for features they do not require. Overly large memory footprints are particularly problematic for embedded systems, such as cellular phones or telecom switch line cards.

Time inefficiency can stem from restricting an ORB to use statically configured algorithms or data structures for key processing tasks. This can make it hard for application developers to customize an ORB to handle new user-cases. For instance, real-time avionics systems [10] often can instantiate all their servants off-line. These systems can benefit from an ORB that uses perfect hashing or active demultiplexing [16] to demultiplex incoming requests to servants. However, ORBs that are configured statically to use a general-purpose, "one-size-fits-all" demultiplex strategy will not perform as well for mission-critical systems.

In theory, the drawbacks with static configuration described above are *internal* to ORBs and should not affect application developers directly. In practice, however, application developers are inevitably affected since the quality, portability, usability, and performance of the ORB middleware is reduced. Therefore, an effective way to improve ORB extensibility is to develop ORB middleware that can be *dynamically configured*.

Dynamic configuration enables the selective integration of customized implementations for key ORB strategies, such as communication, concurrency, demultiplexing, scheduling, and dispatching. This allows ORB developers to concentrate on the *functionality* of ORB components, without committing themselves prematurely to a specific *configuration* of these components. Moreover, dynamic configuration enables application developers and ORB developers to make these decisions very late in the design lifecycle, *i.e.*, at installation-time or run-time.

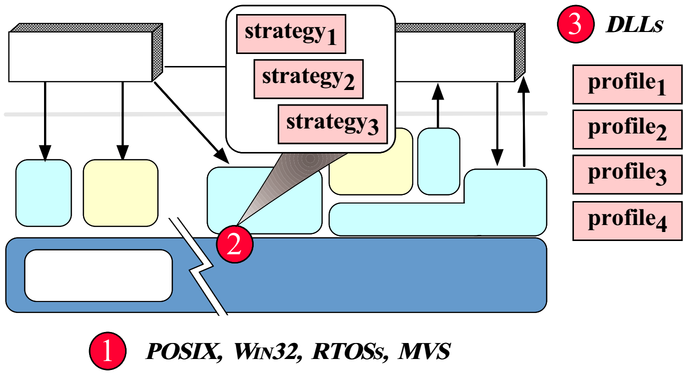

Figure 37: Dimensions of ORB Extensibility

Figure 37 illustrates the following key dimensions of ORB extensibility:

1. Extensibility to retargeting on new platforms: which requires that the ORB be implemented using modular components that shield it from non-portable system mechanisms, such as those for threading, communication, and event demultiplexing. OS platforms like POSIX, Win32, VxWorks, and MVS provide a wide variety of system mechanisms.

2. Extensibility via custom implementation strategies: which can be tailored to specific application requirements. For instance, ORB components can be customized to meet periodic deadlines in real-time systems [10]. Likewise, ORB components can be customized to account for particular system characteristics, such as the availability of asynchronous I/O [16] or high-speed ATM networks [71].

3. Extensibility via dynamic configuration of custom strategies: which takes customization to the next level by dynamically linking only those strategies that are necessary for a specific ORB “personality.” For example, different application domains, such as medical systems or telecom call processing, may require custom combinations of concurrency, scheduling, or dispatch strategies. Configuring these strategies at run-time from dynamically linked libraries (DLLs) can (1) reduce the memory footprint of an ORB and (2) make it possible for application developers to extend the ORB without requiring access or changes to the original source code.

Below, we describe the patterns applied to enhance the extensibility of TAO along each dimension outlined above.

6.2 Overview of Patterns that Improve ORB Extensibility

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. Figure 38 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

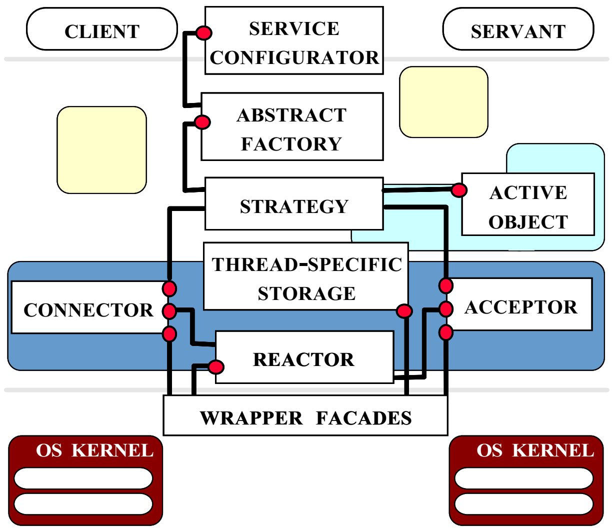

Figure 38: Relationships Among Patterns Used in TAO

detail or to discuss all the patterns used within TAO. Instead, our goal is to focus on key patterns and show how they can improve the extensibility, maintainability, and performance of real-time ORB middleware. The references contain additional material on each pattern.

The intent and usage of these patterns are outlined below:

The Wrapper Facade pattern: which simplifies the OS system programming interface by combining multiple related OS system mechanisms like the socket API or POSIX threads into cohesive OO abstractions [48]. TAO uses this pattern to avoid tedious, non-portable, and non-typesafe programming of low-level, OS-specific system calls.

The Reactor pattern: which provides flexible event demultiplexing and event handler dispatching [43]. TAO uses this pattern to notify ORB-specific handlers synchronously when I/O events occur in the OS. The Reactor pattern drives the main event loop in TAO’s ORB Core, which accepts connections and receives/sends client requests/responses.

The Acceptor-Connector pattern: which decouples GIOP protocol handler initialization from the ORB processing tasks performed once initialization is complete [45]. TAO uses this pattern in the ORB Core on servers and clients to passively and actively establish GIOP connections that are independent of the underlying transport mechanisms.

The Active Object pattern: which supports flexible concurrency architectures by decoupling request reception from request execution [51]. TAO uses this pattern to facilitate the use of multiple concurrency strategies that can be configured flexibly into its ORB Core at run-time.

The Thread-Specific Storage pattern: which allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access [59]. TAO uses this pattern to minimize lock contention and priority inversion for real-time applications.

The Strategy pattern: which provides an abstraction for selecting one of several candidate algorithms and packaging it into an object [48]. This pattern is the foundation of TAO's extensible software architecture and makes it possible to configure custom ORB strategies for concurrency, communication, scheduling, and demultiplexing.

The Abstract Factory pattern: which provides a single factory that builds related objects. TAO uses this pattern to consolidate its dozens of Strategy objects into a manageable number of abstract factories that can be reconfigured *en masse* into clients and servers conveniently and consistently. TAO components use these factories to access related strategies without explicitly specifying their subclass name [48].

The Service Configurator pattern: which permits dynamic run-time configuration of abstract factories and strategies in an ORB [50]. TAO uses this pattern to dynamically interchange abstract factory implementations in order to customize ORB personalities at run-time.

It is important to note that the patterns described in this section are not limited to ORBs or communication middleware. 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.

6.3 How to Use Patterns to Resolve ORB Design Challenges

In the following discussion, we outline the forces that underlie the key design challenges that arise when developing extensible real-time ORBs. We also describe which pattern(s) resolve these forces and explain how these patterns are used in TAO. In addition, we show how the absence of these patterns in an ORB leaves these forces unresolved. To illustrate this latter point concretely, we compare TAO with SunSoft IIOP, which is a freely available[11] reference implementation of the Internet Inter-ORB Protocol (IIOP) written in C++. TAO evolved from the SunSoft IIOP release, so it provides an ideal baseline to evaluate the impact of patterns on the software qualities of ORB middleware.

6.3.1 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 client requests concurrently. Figure 39 illustrates a common approach used by SunSoft IIOP, which is programmed internally using system mechanisms like sockets, `select`, and POSIX threads directly.

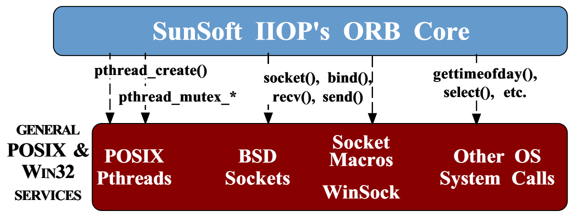

Figure 39: SunSoft IIOP Operating System Interaction

Problem: Developing an ORB is hard. It is even harder if developers must wrestle with low-level system mechanisms written in languages like C, which often yield the following problems:

- **ORB developers must have intimate knowledge of many OS platforms:** Implementing an ORB using system-level C APIs forces developers to deal with non-portable, tedious, and error-prone OS idiosyncrasies, such as using untyped socket handles to identify transport endpoints. Moreover, these APIs are not portable across OS platforms. For example, Win32 lacks POSIX threads 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, as shown in Section 6.5. It is hard to maintain and extend such ORBs since platform-specific details are scattered throughout the implementation source code files.

[11]See `ftp://ftp.omg.org/pub/interop/` for the SunSoft IIOP source code.

• **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 [48]. 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 typesafe, modular, and portable class interfaces that encapsulate lower-level, stand-alone system mechanisms, such as sockets, `select`, and POSIX threads. In general, the Wrapper Facade pattern should be applied when existing system-level APIs are non-portable and non-typesafe.

Using the Wrapper Facade pattern in TAO: TAO accesses all system mechanisms via the wrapper facades provided by ACE [24]. 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 40 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 typesafe OO interfaces. The OO encapsulation provided by ACE alleviates the need for TAO to access the weakly-typed system APIs directly. Therefore, C++ compilers can detect type system violations at compile-time rather than at run-time.

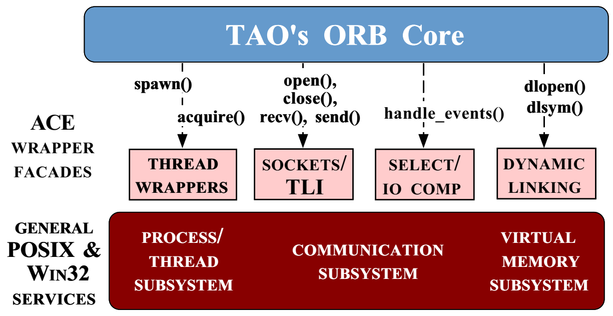

Figure 40: TAO's Wrapper Facade Encapsulation

The 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 short methods. Likewise, static methods are used to avoid the overhead of passing a C++ `this` pointer to each invocation.

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.

6.3.2 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.

Figure 41 illustrates a typical event demultiplexing sequence for SunSoft IIOP. In (**1**), the server enters its event loop by (**2**) calling `get_request` on the Object Adapter. The `get_request` method then (**3**) calls the static method `block_for_connection` on the `server_endpoint`. This method manages all aspects of server-side connection management, ranging from connection establishment to GIOP protocol handling. The ORB remains blocked (**4**) on `select`

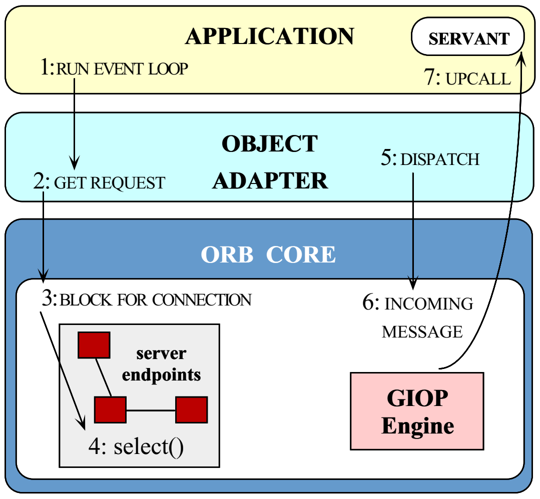

Figure 41: SunSoft IIOP Event Loop

until the occurrence of I/O event, such as a connection event or a request event. When a request event occurs, `block_for_connection` demultiplexes that request to a specific `server_endpoint` and (**5**) dispatches the event to that endpoint. The GIOP Engine in the ORB Core then (**6**) retrieves data from the socket and passes it to the Object Adapter, which demultiplexes it, demarshals it, and (**7**) dispatches the appropriate method upcall to the user-supplied servant.

Problem: One way to develop an ORB Core is to hard-code it to use one event demultiplexing mechanism, such as `select`. 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 [66], whereas synchronous threads are the most efficient demultiplexing mechanism on Solaris [67].

Another way to develop an ORB Core is to tightly couple its event demultiplexing code with the code that performs GIOP protocol processing. For instance, the event demultiplexing logic of SunSoft IIOP is not a self-contained component. Instead, it is closely intertwined with subsequent processing of client request events by the Object Adapter and IDL skeletons. In this case, the demultiplexing code cannot be reused as a blackbox component by similar communication middleware applications, such as HTTP servers [66] 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 decouple itself from 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* [43]. 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 applications or components like an ORB Core must handle events from multiple clients concurrently, without becoming tightly coupled 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 event demultiplexing 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 42. A TAO server (**1**) initiates 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.

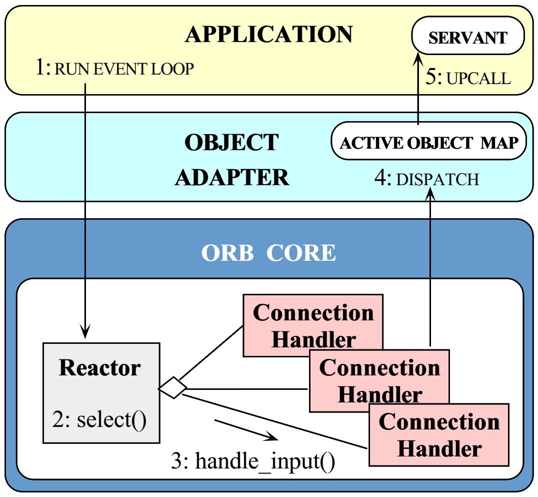

Figure 42: Using the Reactor Pattern in TAO's Event Loop

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 can be used on Windows NT, whereas `select` can be 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 network traffic does not affect the Reactor's implementation. Finally, unlike the event demultiplexing code in SunSoft IIOP, which is tightly coupled to one use-case, the ACE implementation of the Reactor pattern [69] used by TAO has been applied in many other OO event-driven applications ranging from HTTP servers [66] to real-time avionics infrastructure [10].

6.3.3 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 can focus solely on processing client requests, rather than dealing with low-level network programming tasks.

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 can preclude its use as a real-time protocol [23]. Likewise, it may be more efficient to use a shared memory transport mechanism when clients and servants are collocated on the same endsystem. Ideally, an ORB Core should be flexible enough to support multiple transport mechanisms.

Problem: The CORBA architecture explicitly decouples (1) the connection management tasks performed by an ORB Core from (2) the request processing performed by application-specific servants. A common way to implement an ORB's *internal* connection management activities, however, is to use low-level network APIs like sockets. Likewise, the ORB's connection establishment protocol is often tightly coupled with the communication protocol.

Figure 43 illustrates the connection management structure of SunSoft IIOP. The client-side of SunSoft IIOP implements a hard-coded connection caching strategy that uses a linked-list of `client_endpoint` objects. As shown in Figure 43, this list is traversed to find an unused endpoint whenever (**1**) `client_endpoint::lookup` is called. If no unused `client_endpoint` to the server is in the cache, a new connection (**2**) is initiated; otherwise an existing connection is reused. Likewise, the server-side uses a linked list of `server_endpoint` objects to generate the read/write bitmasks required by the (**3**) `select` event demultiplexing mechanism. This list maintains passive transport endpoints that (**4**) accept connections and (**5**) receive requests from clients connected to the server.

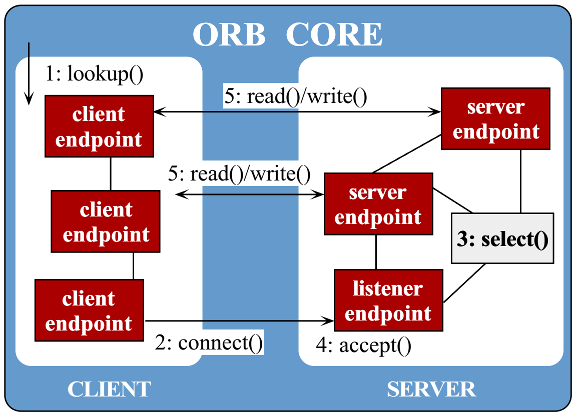

Figure 43: Connection Management in SunSoft IIOP

The problem with this design is that it tightly couples (1) the ORB's connection management implementation with the socket network programming API and (2) the TCP/IP connection establishment protocol with the GIOP communication protocol, yielding the following drawbacks:

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, tightly 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. Thus, it can be hard to port such a tightly coupled ORB Core to new networks, such as ATM or Fibrechannel, 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 software development cycle, *e.g.*, after systematic performance profiling. For example, to reduce lock contention and overhead, a multi-threaded, real-time ORB client may need to store transport endpoints in thread-specific storage [59]. 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, it is hard to accommodate efficient new strategies if connection management mechanisms are hard-coded and tightly bound with other internal ORB 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* [45]. 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 connection establishment for GIOP/IIOP communication. Within TAO's client-side ORB Core, a `Connector` initiates connections to servers in response to an operation invocation or explicit binding to a remote object. Within TAO's server-side ORB Core, an `Acceptor` creates a GIOP `Connection Handler` to service each new client connection. `Acceptors` and `Connection_Handlers` both derive from an `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 select an appropriate concurrency mechanism, as described in Section 6.3.4.

Figure 44 illustrates the use of Acceptor-Connector strategies in TAO's ORB Core. When a client (**1**) invokes a

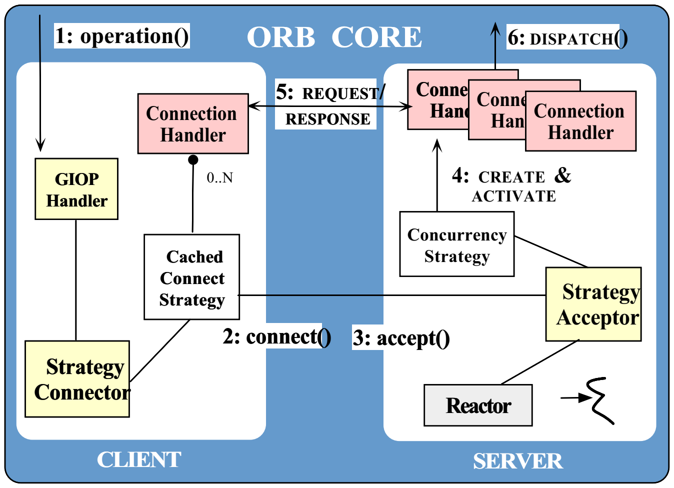

Figure 44: Using the Acceptor-Connector Pattern in TAO's Connection Management

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 and only creates new connections when all existing connections are busy. This caching 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-connection, thread-per-priority, etc., described in Section 6.3.4. 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.

6.3.4 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 directly address the issue of concurrency within an ORB or a servant. Instead, it defines an interface on the POA for an application to specify that all requests be handled by a single thread or be handled using the ORB's internal multi-threading policy. In particular, the POA specification does not allow applications to specify concurrency models, such as thread-per-request or thread pools, which makes it inflexible for certain types of applications [52].

To meet application QoS requirements, it is important to develop ORBs that manage concurrent processing efficiently [42]. 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 [10].

Concurrency is often implemented via the multi-threading capabilities available on OS platforms. For instance, SunSoft IIOP supports the two concurrency architectures shown in Figure 45: a single-threaded Reactive architecture and a thread-per-connection architecture.

SunSoft IIOP's reactive concurrency architecture uses `select` within a single thread to dispatch each arriving request to an individual `server_endpoint` object, which subsequently reads the request from the appropriate OS kernel queue. In (**1**), a request arrives and is queued by the OS. Then, `select` fires, (**2**) notifying the associated `server_endpoint` of a waiting request. The `server_endpoint` finally (**3**) reads the request from the queue and processes it.

In contrast, SunSoft IIOP's thread-per-connection architecture executes each `server_endpoint` in its own thread of control, servicing all requests arriving on that connection within its thread. After a connection is established, `select` waits for events on the connection's descriptor. When (**1**) requests are received by the OS, the thread performing `select` (**2**) reads one from the queue and (**3**) hands it off to a

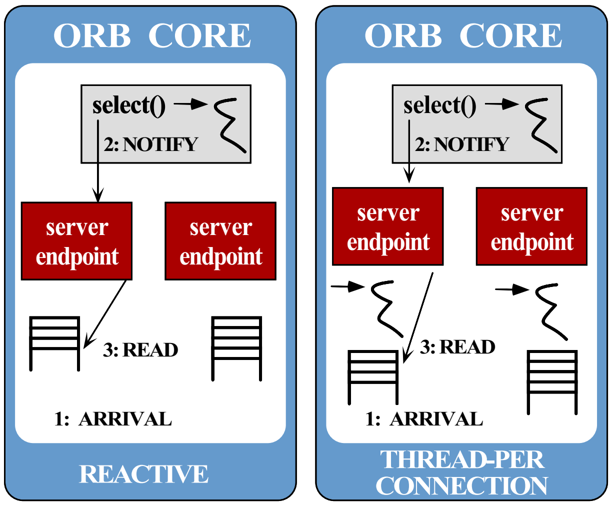

Figure 45: SunSoft IIOP Concurrency Architectures

`server_endpoint` for processing.

Problem: In many ORBs, the concurrency architecture is programmed directly using the OS platform's multi-threading API, such as the POSIX threads API [72]. However, there are several drawbacks to this approach:

• **Non-portable:** Threading APIs are highly platform-specific. Even industry standards, such as POSIX threads, are not available on many widely-used OS platforms, including Win32, VxWorks, and pSoS. Not only is there no direct syntactic mapping between APIs, but there is no clear mapping of semantic functionality either. For instance, POSIX threads supports deferred thread cancellation, whereas Win32 threads do not. Moreover, although Win32 has a thread termination API, the Win32 documentation strongly recommends *not* using it since it does not release all thread resources on exit. Moreover, even POSIX pthread implementations are non-portable since many UNIX vendors support different drafts of the pthreads specification.

• **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 servants. In addition, the techniques required to robustly terminate servants executing 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 [43] is an appropriate strategy for short duration, compute-bound requests on a uni-processor. If these external factors change, however, an ORB's design should be extensible enough to handle alternative concurrency strategies, such as thread pool or thread-per-priority.

When ORBs are developed using low-level threading APIs, however, they are hard to extend with new concurrency strategies *without* affecting other ORB components. For example, adding a thread-per-request architecture to SunSoft IIOP would require extensive changes in order to (1) store the request in a thread-specific storage (TSS) variable during protocol processing, (2) pass the key to the TSS variable through the scheduling and demarshaling steps in the Object Adapter, and (3) access the request stored in TSS before dispatching the operation on the servant. Therefore, there is no easy way to modify SunSoft IIOP's concurrency architecture without drastically changing its internal structure.

How then can an ORB support a simple, extensible, and portable concurrency mechanism?

Solution $\rightarrow$ 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* [51]. This pattern provides a higher-level concurrency architecture that decouples (1) the thread that initially receives and processes a client request from (2) the thread that ultimately executes this request and/or subsequent requests.

While *Wrapper Facades* provide the basis for portability, they are simply a thin syntactic veneer over the low-level system APIs. Moreover, a facade's semantic 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 flexibility 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 46.

The processing shown in Figure 46 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 single-threading, thread-per-connection, or thread pool, 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 class called `Task`. To process a request

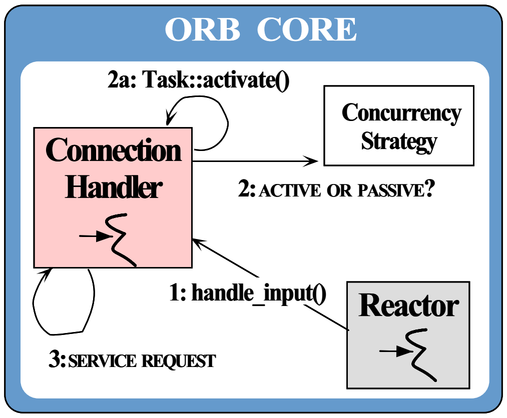

Figure 46: Using the Active Object Pattern to Structure TAO's Concurrency Strategies

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 ultimately (**3**) processes the request.

6.3.5 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. In an ORB, common shared resources include the dynamic memory heap, an object reference created by the `CORBA::ORB_init` ORB initialization factory, the Active Object Map in a POA [73], and the `Acceptor`, `Connector`, and `Reactor` components described earlier.

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, locking overhead negates the 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 [44]. In addition, multi-threaded ORBs are hard to program due to complex concurrency control protocols required to avoid race conditions and deadlocks.

Solution $\rightarrow$ 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 [59]. 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 uses thread-specific storage to store its ORB Core and Object Adapter components, *e.g.*, `Reactor`, `Acceptor`, `Connector`, and `POA`. 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 47. Thus, no additional locking is required to access ORB state.

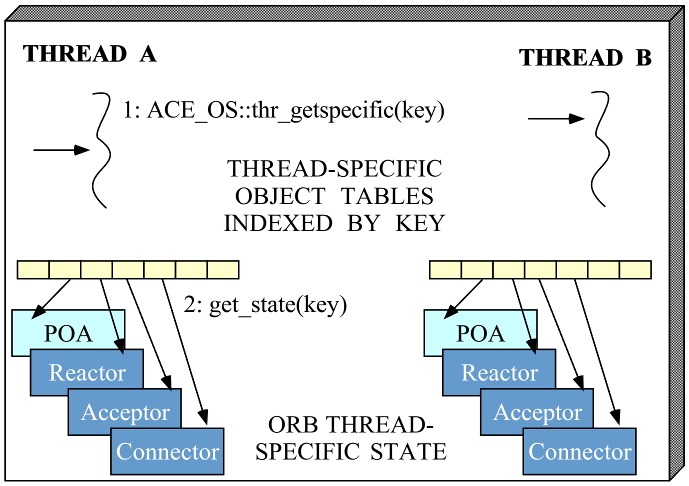
THREAD A
THREAD B
1: ACE_OS::thr_getspecific(key)
THREAD-SPECIFIC OBJECT TABLES INDEXED BY KEY
2: get_state(key)
POA
Reactor
Acceptor
Connector
ORB THREAD-SPECIFIC STATE
POA
Reactor
Acceptor
Connector

Figure 47: Using the Thread-Specific Storage Pattern TAO

6.3.6 Support Interchangeable ORB Behaviors with the Strategy Pattern

Context: The alternative concurrency architectures described in 6.3.4 are just one of the many strategies that an extensible ORB may need to support. In general, extensible ORBs must support multiple request demultiplexing and scheduling strategies in their Object Adapters. Likewise, they must support 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 not available on all OS platforms, conditional compilation is often used to select a feasible 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 for platforms that support multi-threading [42].

While these two configuration approaches are widely used, they are 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. Such 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* [48]. 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 via multiple strategies that can be interchanged seamlessly.

Using the Strategy Pattern in TAO: TAO uses a variety of strategies to factor out behaviors that are typically hard-coded in conventional ORBs. Several of these strategies are illustrated in Figure 48. For instance, TAO supports multiple request demultiplexing strategies (*e.g.*, perfect hashing vs. active demultiplexing [16]) and scheduling strategies (*i.e.*, FIFO vs. rate monotonic vs. maximal urgency first [46]) in its Object Adapter, as well as connection management strategies (*e.g.*, process-wide cached connections vs. thread-specific cached connections) and handler concurrency strategies (*e.g.*, Reactive vs. variations of Active Objects) in its ORB Core.

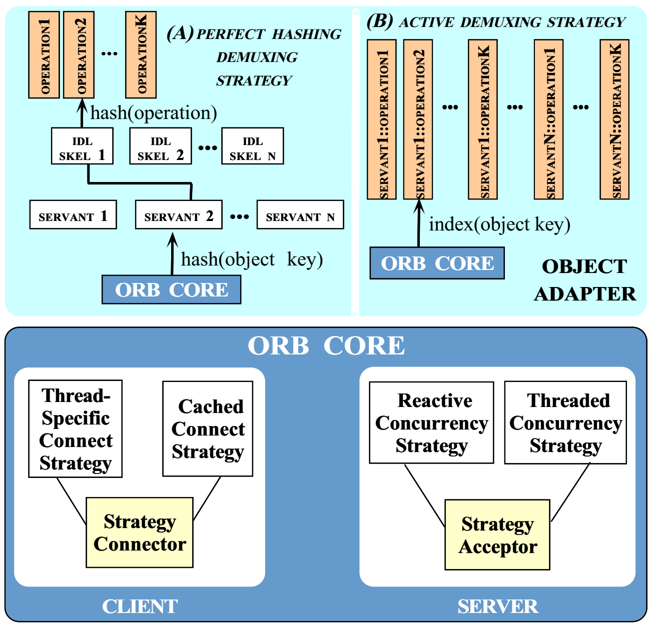

Figure 48: Strategies in TAO

6.3.7 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. Configuration 1 is an avionics application with deterministic real-time requirements [10]. Configuration 2 is an electronic medical imaging application [11] 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:

• **Complicated maintenance and configuration:** ORB source code can become littered with hard-coded references to strategy types, which complicates maintenance and configuration. For example, within a particular application domain, such as real-time avionics or medical imaging, many independent strategies must act in harmony. Identifying these strategies individually by name, however, 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

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

Table 1: Example Applications and their ORB Strategy Configurations

may 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 vs. 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 [20].

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 $\rightarrow$ the Abstract Factory pattern: An effective way to consolidate multiple ORB strategies into semantically compatible configurations is to apply the *Abstract Factory pattern* [48]. 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 *en masse* in semantically meaningful ways. In general, the Abstract Factory pattern should be used when an application must consolidate the configuration of many strategies, each having multiple alternatives that must vary together.

Using the Abstract Factory pattern in TAO: All of TAO's ORB strategies are consolidated into two abstract factories that are implemented as Singletons [48]. One factory encapsulates client-specific strategies, the other factory encapsulates server-specific strategies, as shown in Figure 49. These abstract factories encapsulate request demultiplexing, scheduling, and dispatch strategies in the server, as well as concurrency strategies in both client and server. By using the Abstract Factory pattern, TAO can configure different ORB personalities conveniently and consistently.

6.3.8 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.

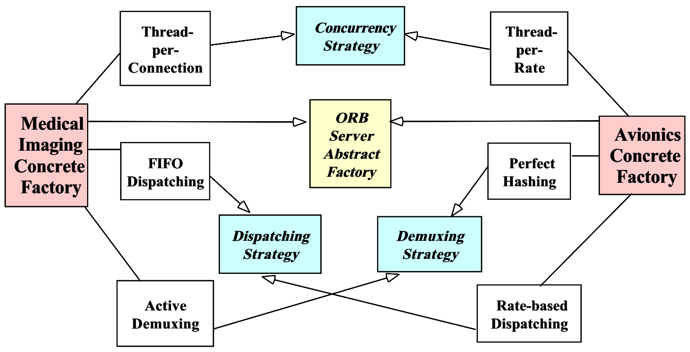

Figure 49: Factories used in TAO

This parsimony is particularly essential for embedded real-time systems that require small memory footprints and predictable CPU processing overhead [74]. Likewise, many applications can benefit from the ability to extend ORBs *dynamically*, *i.e.*, by allowing their strategies to be configured at run-time.

Problem: Although the Strategy and Abstract Factory patterns simplify the customization of 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.

Although it does not use the Strategy pattern explicitly, SunSoft IIOP does permit applications to vary certain ORB strategies at run-time. However, the different strategies must be configured statically into SunSoft IIOP at compile-time. Moreover, as the number of alternatives increases, so does the amount of code required to implement them. For instance, Figure 50 illustrates SunSoft IIOP's approach to varying the concurrency strategy.

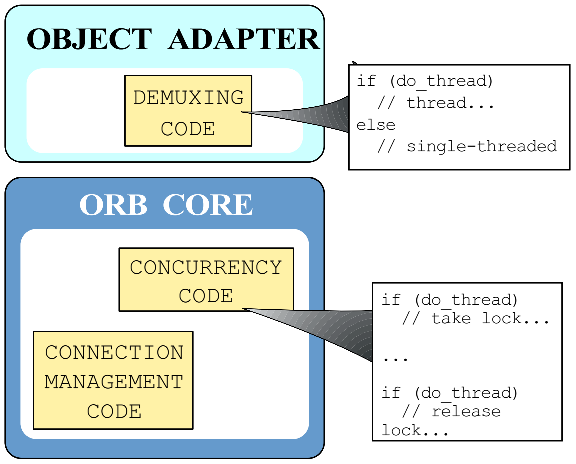

Figure 50: SunSoft IIOP Hard-coded Strategy Usage

Each area of code that might be affected by the choice of concurrency strategy is trusted to act independently of other areas. This proliferation of decision points adversely increases the complexity of the code, complicating future enhancement and maintenance. Moreover, the selection of the data type specifying the strategy complicates integration of new concurrency architectures because the type (`bool`) would have to change, as well as the programmatic structure, `if (do_thread) then ... else ...`, that decodes the strategy specifier into actions.

In general, static configuration is only feasible for a small, fixed number of strategies. However, using this technique to configure complex middleware like 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 → the Service Configurator pattern: An effective way to enhance the dynamism of an ORB is to apply the *Service Configurator pattern* [50]. This pattern uses explicit dynamic linking [70] 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 and/or run-time. Widely available explicit dynamic linking mechanisms include the `dlopen/dlsym/dlclose` functions in SVR4 UNIX [75] and the `LoadLibrary/GetProcAddress` functions in the WIN32 subsystem of Windows NT [76]. The ACE wrapper facades portably encapsulate these OS APIs.

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 and/or system administrators 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 in conjunction with the Strategy and Abstract Factory patterns to dynamically install the strategies it requires without (1) recompiling or statically relinking existing code or (2) terminating and restarting an existing ORB and its application servants. This design allows the behavior of TAO to be tailored for specific platforms and application requirements without requiring access to, or modification of, ORB source code.

In addition, the Service Configurator pattern allows applications to customize the personality of TAO at run-time. For instance, during TAO's ORB initialization phase, it uses the dynamic linking mechanisms provided by the OS, and encapsulated by the ACE wrapper facades, to link in the appropriate concrete factory for a particular use-case. Figure 51 shows two factories tuned for different application domains supported by TAO: avionics and medical imaging.

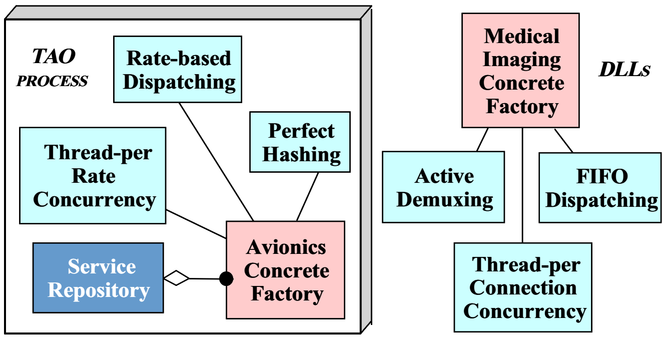
TAO PROCESS
Rate-based Dispatching
Perfect Hashing
Thread-per Rate Concurrency
Service Repository
Avionics Concrete Factory
Medical Imaging Concrete Factory
DLLs
Active Demuxing
FIFO Dispatching
Thread-per Connection Concurrency

Figure 51: Using the Service Configurator Pattern in TAO

In particular configuration shown in Figure 51, the avionics concrete factory has been installed in the process. Applications using this ORB personality will be configured with a particular set of ORB concurrency, demultiplexing, and dispatching strategies. The medical imaging concrete factory resides in a DLL outside of the existing ORB process. To configure a different ORB personality, this factory could be installed dynamically during the ORB server's initialization phase.

6.4 Summary of Design Challenges and Patterns That Resolve Them

Table 2 summarizes the mapping between ORB design challenges and the patterns we applied to resolve these challenges in TAO. This table focuses on the forces resolved by individual patterns. However, TAO also benefits from the collaborations among *multiple* patterns (shown in Figure 38). For example, the Acceptor and Connector patterns utilize the Reactor pattern to notify them when connection events occur at the OS level.

Forces	**Resolving Pattern(s)**
Abstracting low-level system calls	Wrapper Facade
ORB event demultiplexing	Reactor
ORB connection management	Acceptor, Connector
Efficient concurrency models	Active Object
Pluggable strategies	Strategy
Group similar initializations	Abstract Factory
Dynamic run-time configuration	Service Configurator

Table 2: Summary of Forces and their Resolving Patterns

Moreover, patterns often must collaborate to alleviate drawbacks that arise from applying them in isolation. For instance, the reason the Abstract Factory pattern is used in TAO is to avoid the complexity caused by its extensive use of the Strategy pattern. Although the Strategy pattern simplifies the effort required to customize an ORB for specific application requirements and network/endsystem characteristics, it is tedious and error-prone to manage a large number of strategy interactions manually.

6.5 Evaluating the Contribution of Patterns to ORB Middleware

Section 6.3 described the key patterns used in TAO and qualitatively evaluated how these patterns helped to alleviate limitations with the design of SunSoft IIOP. The discussion below goes one step further and quantitatively evaluates the benefits of applying patterns to ORB middleware.

6.5.1 Where's the Proof?

Implementing TAO using patterns yielded significant quantifiable improvements in software reusability and maintainability. The results are summarized in Table 3. This table compares the following metrics for TAO and SunSoft IIOP:

1. The number of methods required to implement key ORB tasks (such as connection management, request transfer, socket and request demultiplexing, marshaling, and dispatching).
2. The total non-comment lines of code (LOC) for these methods.
3. The average McCabe Cyclometric Complexity metric $v(G)$ [77] of the methods. The $v(G)$ metric uses graph theory to correlate code complexity with the number of possible basic paths that can be taken through a code module. In C++, a module is defined as a method.

The use of patterns in TAO significantly reduced the amount of *ad hoc* code and the complexity of certain operations. For instance, the total lines of code in the client-side *Connection Management* operations were reduced by a factor of 5. Moreover, the complexity for this component was substantially reduced by a factor of 16. These reductions in LOC and complexity stem from the following factors:

- These ORB tasks were the focus of our initial work when developing TAO.
- Many of the details of connection management and socket demultiplexing were subsumed by patterns and components in the ACE framework, in particular, the Acceptor, Connector, and Reactor.

Other areas did not yield as much improvement. In particular, *GIOP Invocation* tasks actually increased in size and maintained a consistent $v(G)$. There were two reasons for this increase:

1. The primary pattern applied in these cases was the Wrapper Facade, which replaced the low-level system calls with ACE wrappers but did not factor out common strategies; and
2. SunSoft IIOP did not trap all the error conditions, which TAO addressed much more completely. Therefore, the additional code in TAO is necessary to provide a more robust ORB.

The most compelling evidence that the systematic application of patterns can positively contribute to the maintainability of complex software is shown in Figure 52. This figure illustrates the distribution of $v(G)$ over the percentage of affected methods in TAO. As shown in the figure, most of TAO's code is structured in a straightforward manner, with almost 70% of the methods' $v(G)$ falling into the range of 1-5.

In contrast, while SunSoft IIOP has a substantial percentage (55%) of its methods in that range, many of the remaining methods (29%) have $v(G)$ greater than 10. The reason for the difference is that SunSoft IIOP uses a monolithic coding style

	TAO			SunSoft IIOP		
ORB Task	*# Methods*	*Total LOC*	*Avg.* $v(G)$	*# Methods*	*Total LOC*	*Avg.* $v(G)$
Connection Management (Server)	2	43	7	3	190	14
Connection Management (client)	3	11	1	1	64	16
GIOP Message Send (client/Server)	1	46	12	1	43	12
GIOP Message Read (client/Server)	1	67	19	1	56	18
GIOP Invocation (client)	2	205	26	2	188	27
GIOP Message Processing (client/Server)	3	41	2	1	151	24
Object Adapter Message Dispatch (Server)	2	79	6	1	61	10

Table 3: Code Statistics: TAO vs. SunSoft IIOP

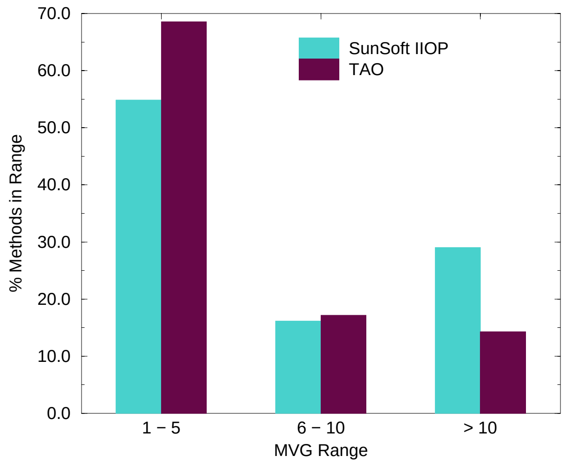

Figure 52: Distribution of $v(G)$ Over ORB Methods

with long methods. For example, the average length of methods with $v(G)$ over 10 is over 80 LOC. This yields overly-complex code that is hard to debug and understand.

In TAO, most of the monolithic SunSoft IIOP methods were decomposed into smaller methods when integrating the patterns. The majority (86%) of TAO's methods have $v(G)$ under 10. Of that number, nearly 70% have a $v(G)$ between 1 and 5. The relatively few (14%) methods in TAO with $v(G)$ greater than 10 are largely unchanged from the original SunSoft IIOP TypeCode interpreter.

The use of monolithic methods not only increases the effort of maintaining TAO, it also degrades its performance due to reduced processor cache hits [18]. Therefore, we plan to experiment with the application of other patterns, such as *Command* and *Template Method* [48], to simplify and optimize these monolithic methods into smaller, more cohesive methods. There are a few methods with $v(G)$ greater than 10 which are not part of the TypeCode interpreter, and they will likely remain that way. Sometimes solving complex problems involves writing complex code; at such times, localizing complexity is a reasonable recourse.

6.5.2 What are the Benefits?

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

Increased extensibility: Patterns like Abstract Factory, Strategy, and Service Configurator simplify the configure of TAO for a particular application domain by allowing extensibility to be "designed into" the ORB. In contrast, middleware that lacks these patterns is significantly harder to extend. This article illustrated how patterns were applied to make the TAO ORB more extensible.

Enhanced design clarity: By applying patterns to TAO, not only did we develop a more flexible ORB, we also devised a richer vocabulary for expressing ORB middleware designs. In particular, patterns capture and articulate the design rationale for complex object-structures in an ORB. Moreover, patterns help to demystify and motivate the structure of an ORB by describing its architecture in terms of design forces that recur in many types of software systems. The expressive power of patterns enabled us to concisely convey the design of complex software systems like TAO. As we continue to learn about ORBs and the patterns of which they are composed, we expect our pattern vocabulary to grow and evolve.

Thus, the patterns presented in this article help to improve the maintainability of ORB middleware by reducing software complexity, as shown in Figure 52.

Increased portability and reuse: TAO is built atop the ACE framework, which provides implementations of many key communication software patterns [9]. Using ACE simplified the porting of TAO to numerous OS platforms since most of the porting effort was 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. This is indicated by the consistent decrease in lines of code (LOC) in Table 3.

6.5.3 What are the Liabilities?

The use of patterns can also 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 [18] and demultiplexing overhead [16]) to minimize performance overhead. As a result, TAO is substantially faster than the original hard-coded SunSoft IIOP [18].

Additional external dependencies: Whereas SunSoft IIOP only depends on system-level interfaces and libraries, TAO 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.

7 Concluding Remarks

Advances in distributed object computing technology are occurring at a time when deregulation and global competition are motivating the need for increased software productivity and quality. Distributed object computing is a promising paradigm to control costs through open systems and client/server computing. Likewise, OO design and programming are widely touted as an effective means to reduce software cost and improve software quality through reuse, extensibility, and modularity.

Meeting the QoS requirements of high-performance and real-time applications requires more than OO design and programming techniques, however. It requires an integrated architecture that delivers end-to-end QoS guarantees at multiple levels of a distributed system. The TAO ORB endsystem described in this paper addresses this need with policies and mechanisms that span network adapters, operating systems, communication protocols, and ORB middleware.

We believe the future of real-time ORBs is very promising. Real-time system development strategies will migrate towards those used for "mainstream" systems to achieve lower development cost and faster time to market. We have observed real-time embedded software development projects that have lagged in terms of design and development methodologies (and languages) by *decades*. These projects are extremely costly to evolve and maintain. Moreover, they are so specialized that they cannot be adapted to meet new market opportunities.

The flexibility and adaptability offered by CORBA make it very attractive for use in real-time systems. If the real-time challenges can be overcome, and the progress reported in this paper indicates that they can, then the use of Real-time CORBA is compelling. Moreover, the solutions to these challenges will sufficiently complex, yet general, that it will be well worth re-applying them to other projects in domains with stringent QoS requirements.

The C++ source code for TAO and ACE is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. This release also contains the real-time ORB benchmarking test suite described in Section 5.3.

Acknowledgments

We gratefully acknowledge Expersoft, IONA, and Sun for providing us with their ORB software for the benchmarking testbed. We would also like to thank Frank Buschmann for his extensive comments on this paper. Finally, we are deeply appreciative of the efforts Chris Gill, Andy Gokhale, Irfan Pyarali, Carlos O'Ryan, and Nanbor Wang put into developing and optimizing TAO's real-time ORB Core and Scheduling Service.

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] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

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

[6] 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.

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

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

[9] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[10] 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.

[11] 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.

[12] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

[13] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Proceedings of the Real-Time Systems Symposium*, (Huntsville, Alabama), December 1988.

[14] S. Khanna and et. al., "Realtime Scheduling in SunOS 5.0," in *Proceedings of the USENIX Winter Conference*, pp. 375–390, USENIX Association, 1992.

[15] V. Fay-Wolfe, J. K. Black, B. Thuraisingham, and P. Krupp, "Real-time Method Invocations in Distributed Environments," Tech. Rep. 95-244, University of Rhode Island, Department of Computer Science and Statistics, 1995.

[16] 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.

[17] D. C. Schmidt, R. Bector, D. Levine, S. Mungee, and G. Parulkar, "An ORB Endsystem Architecture for Statically Scheduled Real-time Applications," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

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

[19] 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.

[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] 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.

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

[23] 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.

[24] 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.

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

[26] A. Gokhale and D. C. Schmidt, "Design Principles and Optimizations for High-performance ORBs," in 12^{th} OOPSLA Conference, poster session, (Atlanta, Georgia), ACM, October 1997.

[27] C. Cranor and G. Parulkar, "Design of Universal Continuous Media I/O," in *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '95)*, (Durham, New Hampshire), pp. 83–86, Apr. 1995.

[28] R. Gopalakrishnan and G. Parulkar, "Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing," in *SIGMETRICS Conference*, (Philadelphia, PA), ACM, May 1996.

[29] 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.

[30] R. Gopalakrishnan and G. M. Parulkar, "Efficient User Space Protocol Implementations with QoS Guarantees using Real-time Upcalls," Tech. Rep. 96-11, Washington University Department of Computer Science, March 1996.

[31] P. Hoschka, "Automating Performance Optimization by Heuristic Analysis of a Formal Specification," in *Proceedings of Joint Conference for Formal Description Techniques (FORTE) and Protocol Specification, Testing and Verification (PSTV)*, (Kaiserslautern), 1996.

[32] P. Druschel, M. B. Abbott, M. Pagels, and L. L. Peterson, "Network subsystem design," *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, vol. 7, July 1993.

[33] J. A. Stankovic, "Misconceptions About Real-Time Computing," *IEEE Computer*, vol. 21, pp. 10–19, Oct. 1988.

[34] R. Braden et al, "Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification." Internet Draft, May 1997. ftp://ietf.org/internet-drafts/draft-ietf-rsvp-spec-15.txt.

[35] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[36] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[37] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1993.

[38] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

[39] V. Kalogeraki, P. Melliar-Smith, and L. Moser, "Soft Real-Time Resource Management in CORBA Distributed Systems," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[40] 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.

[41] Object Management Group, *Realtime CORBA 1.0 Request for Proposals*, OMG Document orbos/97-09-31 ed., September 1997.

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

[43] 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.

[44] 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.

[45] 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.

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

[47] G. Parulkar, D. C. Schmidt, and J. S. Turner, "$\mathrm{a^It^Pm}$: a Strategy for Integrating IP with ATM," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.

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

[49] D. B. Stewart and P. K. Khosla, "Real-Time Scheduling of Sensor-Based Control Systems," in *Real-Time Programming* (W. Halang and K. Ramamritham, eds.), Tarrytown, NY: Pergamon Press, 1992.

[50] 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.

[51] 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.

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

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

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

[55] 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.

[56] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[57] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic Construction of Sparse Data Flow Evaluation Graphs," in *Conference Record of the Eighteenth Annual ACE Symposium on Principles of Programming Languages*, ACM, January 1991.

[58] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," in *ACM Transactions on Programming Languages and Systems*, ACM, October 1991.

[59] 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.

[60] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, vol. 38, pp. 1110–1123, Aug. 1989.

[61] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[62] P. S. Inc., *Quantify User's Guide*. PureAtria Software Inc., 1996.

[63] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[64] M. Guillemont, "CHORUS/ClassiX r3 Technical Overview (technical report #CS/TR-96-119.13)," tech. rep., Chorus Systems, May 1997.

[65] Object Management Group, *Minimum CORBA - Request for Proposal*, OMG Document orbos/97-06-14 ed., June 1997.

[66] 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.

[67] 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.

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

[69] 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.

[70] 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.

[71] 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.

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

[73] I. Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter," *ACM StandardView*, vol. 6, Mar. 1998.

[74] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *submitted to the Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 1998.

[75] 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.

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

[77] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, Dec. 1976.