The Design of the TAO Real-Time Object Request Broker

Douglas C. Schmidt, David L. Levine, and Sumedh Mungee
{schmidt,levine,sumedh}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA[*]

January 18, 1999

This article appeared in Computer Communications, Elsivier Science, Volume 21, No 4, April, 1998.

Abstract

Many real-time application domains can benefit from flexible and open distributed architectures, such as those defined by the CORBA specification. CORBA is an architecture for distributed object computing being standardized by the OMG. Although CORBA is well-suited for conventional request/response applications, CORBA implementations are not yet suited for real-time applications due to the lack of key quality of service (QoS) features and performance optimizations.

This paper makes three contributions to the design of real-time CORBA systems. First, the paper describes the design of TAO, which is our high-performance, real-time CORBA-compliant implementation that runs on a range of OS platforms with real-time features including VxWorks, Chorus, Solaris 2.x, and Windows NT. Second, it presents TAO's real-time scheduling service that can provide QoS guarantees for deterministic real-time CORBA applications. Finally, the paper presents performance measurements that demonstrate the effects of priority inversion and non-determinism in conventional CORBA implementations and how these hazards are avoided in TAO.

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.

1.1 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 either 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.

[*]This work was supported in part by NSF grant NCR-9628218, DARPA contract 9701516, Boeing, Siemens, and Sprint.

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:

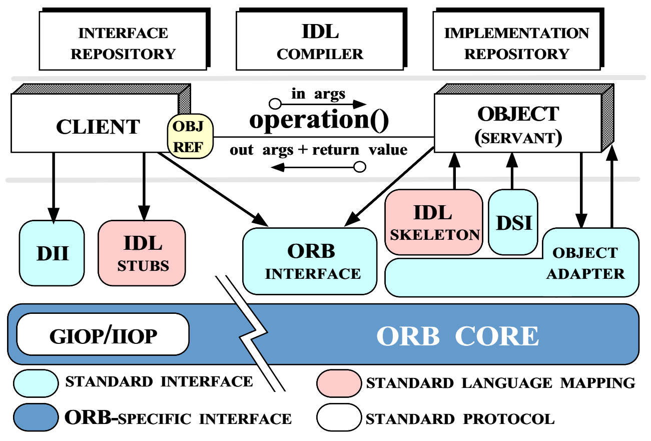

Figure 1: Components in the CORBA 2.x Reference Model

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

Object: In CORBA, an object is an instance of an Interface Definition Language (IDL) interface. The object is identified by an *object reference*, which uniquely names that instance across servers. An *ObjectId* associates an object with its servant implementation, and is unique within the scope of an Object Adapter. 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 `struct`s. A client never interacts with a servant directly, but always through an object identified by an object reference.

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) that 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 to initialize and shutdown the ORB, convert object references to strings and back, and 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 transforms OMG IDL definitions into stubs and skeletons that are generated automatically in 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, which 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 two-way operations to avoid blocking the client until the servant responds. In contrast, in CORBA 2.x, SII stubs only support *two-way*, *i.e.*, request/response, and *one-way*, *i.e.*, request-only operations.[1]

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

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 servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a 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 using the DII. 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.

1.2 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 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 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)

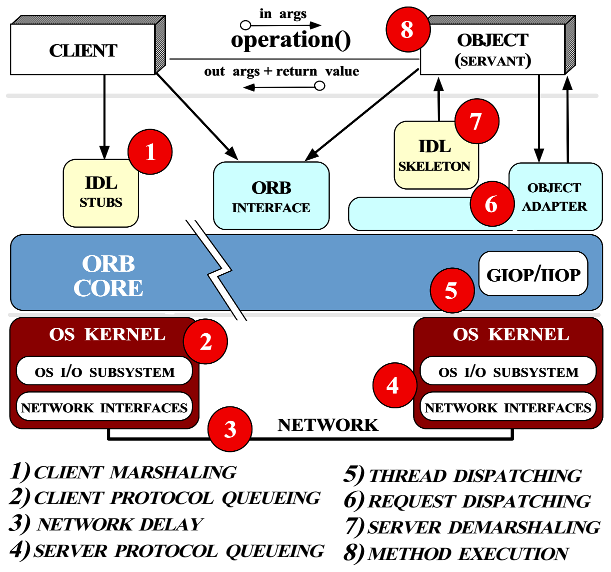

Figure 2: Sources of Latency and Priority Inversion in Conventional ORBs

lack of integration with underlying real-time OS and network QoS mechanisms [22, 23, 17].

1.3 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 2 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.

1.3.1 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.

1.3.2 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.

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

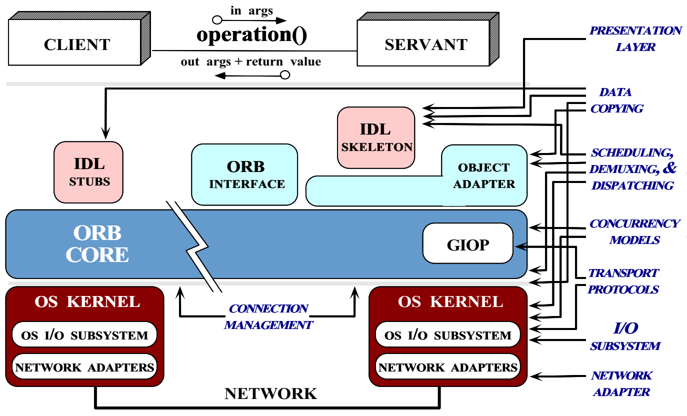

Figure 3: Features and Optimizations for Real-time ORB Endsystems

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 3.3 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 2.1 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 2.2.1 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 2.3 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 2.4 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 non-determinism 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 2.5 outlines TAO's vertically integrated memory management scheme that minimizes data copying and lock contention throughout its ORB endsystem.

1.3.3 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 2.3 can increase ORB performance *and* predictability by eliminating excessive searching and avoiding priority inversions across demultiplexing layers [20].

The remainder of this article is organized as follows: Section 2 describes the feature enhancements and optimizations we are developing for TAO; Section 3 discusses the design and implementation of TAO's real-time Scheduling Service in detail; Section 4 presents performance measurements that demonstrate TAO's ability to support real-time QoS requirements; Section **??** compares our work with related research projects; and Section 5 presents concluding remarks.

2 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 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

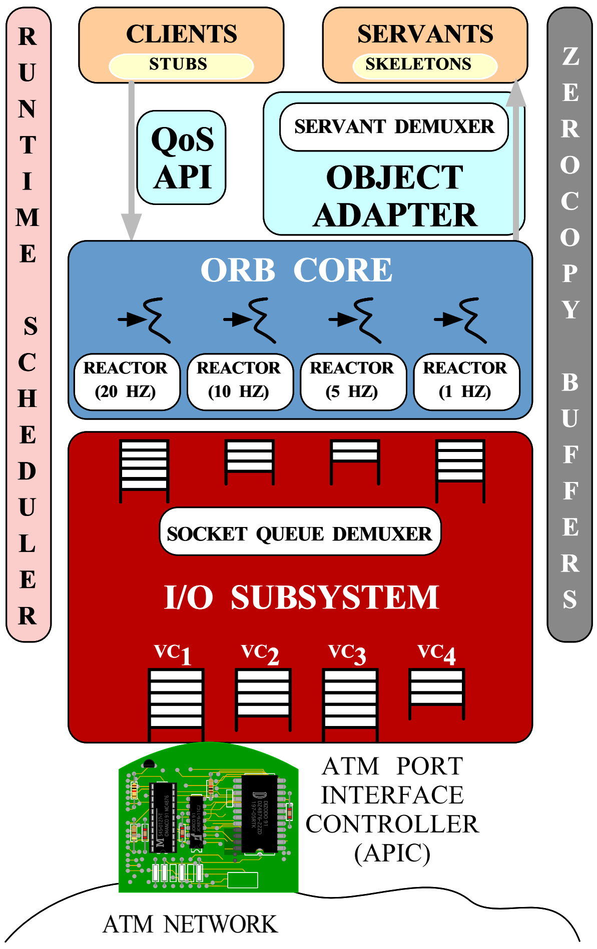

Figure 4: Architectural Components in the TAO Real-time ORB Endsystem

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 1.3. Section 3 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.

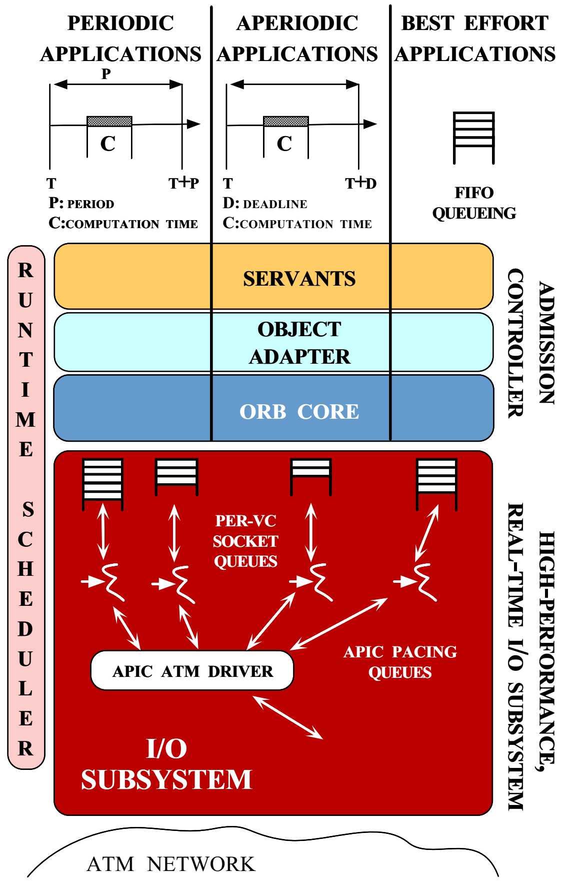

Figure 5: Components in TAO's High-performance, Real-time I/O Subsystem

2.1 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 compactPCI 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.

[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.

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 3.2 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, admission 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.

2.2 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.

2.2.1 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 2.1. 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 2.1 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.

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 with one of the following two models [44]:

- *Client propagation model* – The operation is run at the priority of the client that invoked the operation.
- *Server sets model* – The operation is run at the priority of the thread in the server's ORB Core that received the operation.

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

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

The server sets priority model is well-suited for deterministic real-time applications since it minimizes priority inversion and non-determinism in TAO's ORB Core [45]. 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, avionics 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

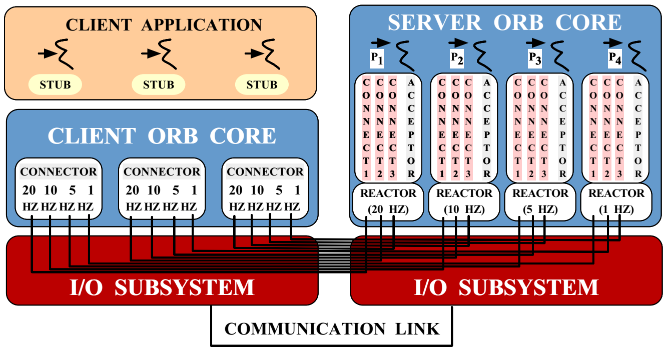

Figure 6: TAO's Priority-based Connection and Concurrency Architectures

thread maintains a `Connector` [46] 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 [45].

Figure 6 also shows how the `Reactor` that is associated with each thread priority in a server ORB can be configured to use an `Acceptor` [46]. 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 [47], 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 **??**, 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 [48]. 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.

2.2.2 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 collocated 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 **??** 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 [49] to allow the configuration of multiple scheduling algorithms, such as earliest deadline first or maximum urgency first [50]. Likewise, the *Bridge* pattern [49] shields TAO's ORB Core from the choice of scheduling algorithm. TAO uses ACE components based on the *Service Configurator* pattern [51] 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 [46] 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 [52] 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.

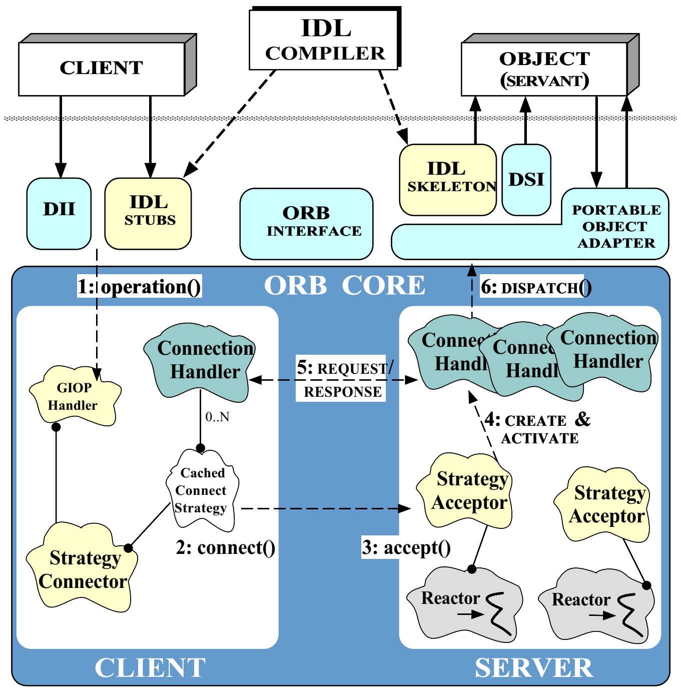

Figure 7: Components in the TAO's ORB Core

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 2.2.1) to the server's `Connection_Handler`. This handler upcalls TAO's Object Adapter, which dispatches the requests to the appropriate servant operation.

2.2.3 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 3. 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 [47].

2.3 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 [53, 54].

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.

2.3.1 Conventional ORB Demultiplexing Strategies

A standard GIOP-compliant client request contains the identity of its object and operation. An object is identified by an object key which is an `octet sequence`. An operation is represented as a `string`. As shown in Figure 8, the ORB

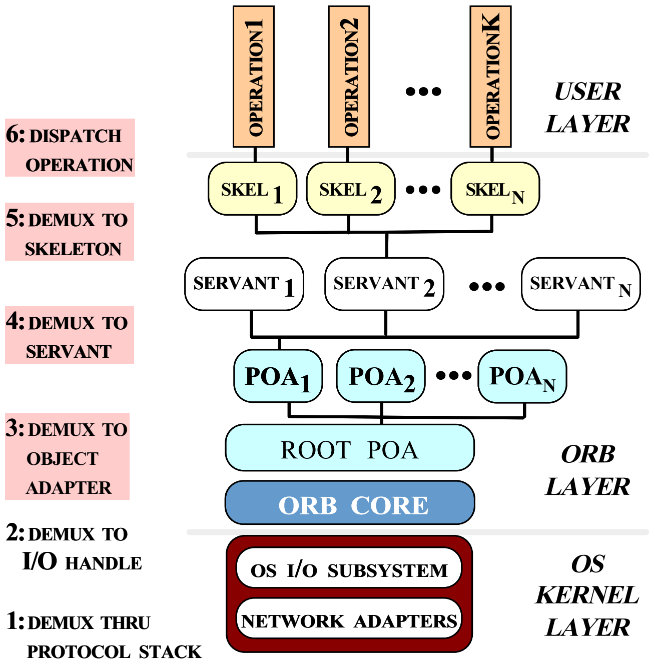

Figure 8: CORBA 2.2 Logical Server Architecture

endsystem must perform the following demultiplexing tasks:

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

Steps 3, and 4: The ORB Core uses the addressing information in the client's object key to locate the appropriate POA and servant. POAs can be organized hierarchically. Therefore, locating the POA that contains the servant can involve multiple demultiplexing steps through the POA hierarchy.

Step 5 and 6: The POA uses the operation name to find the appropriate IDL skeleton, which demarshals the request buffer into operation parameters and performs the upcall to code supplied by servant developers.

The conventional layered ORB endsystem demultiplexing implementation shown in Figure 8 is generally inappropriate for high-performance and real-time applications for the following reasons [55]:

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.

2.3.2 TAO's Optimized ORB Demultiplexing Strategies

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

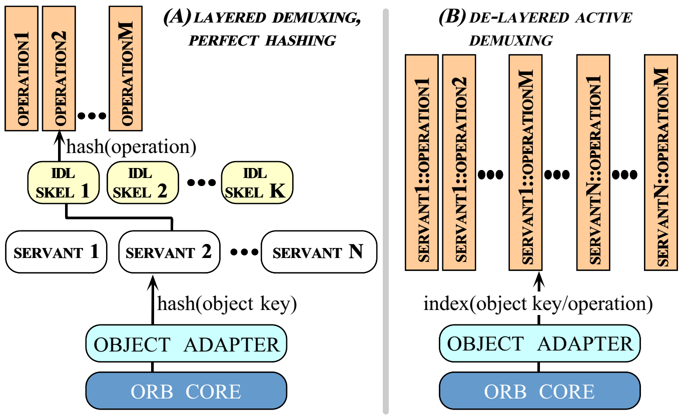

Figure 9: Optimized CORBA Request Demultiplexing Strategies

optimized demultiplexing strategies include the following:

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` [56] 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 avionics mission control systems [10, 47], 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 [51] 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 2.1 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.

2.4 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 [57] 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 [58, 59] 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.

2.5 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 [45]. 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 2.1 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 [60] 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.

3 Supporting Real-time Scheduling in CORBA

Section 2 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.

3.1 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 [47], 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 paper describes the static scheduling service we initially developed for TAO. It then follows the progression of our scheduling research towards dynamic scheduling, for both deterministic and statistical real-time systems.

3.2 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 [47]. 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 critical operations, as discussed in Section 3.5.

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 3.4.

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 3.3.3. 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

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

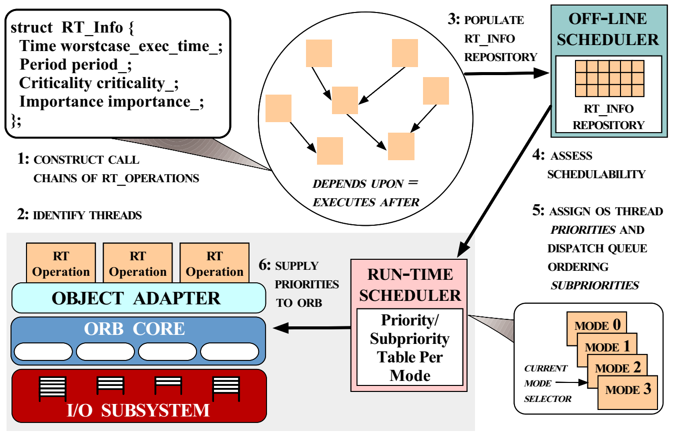
struct RT_Info {
Time worstcase_exec_time_;
Period period_;
Criticality criticality_;
Importance importance_;
};
1: CONSTRUCT CALL CHAINS OF RT_OPERATIONS
2: IDENTIFY THREADS
DEPENDS UPON = EXECUTES AFTER
3: POPULATE RT_INFO REPOSITORY
OFF-LINE SCHEDULER
RT_INFO REPOSITORY
4: ASSESS SCHEDULABILITY
5: 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

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 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.

It is desirable to schedule operations that are not part of the critical set if the dynamic behavior of the system results in additional available CPU resources, but scheduling of a non-critical operation should *never* result in an operation from the critical set failing to execute before deadline.

To simplify the presentation, we focus on ORB scheduling for a single CPU. The distributed scheduling problem is not addressed in this presentation. [47] outlines the approaches we are investigating with TAO.

3.3 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.

3.3.1 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 [61]. 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.

3.3.2 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]

[6]The remainder of the `RT_Scheduler` module IDL description is shown in Section 3.5.1.

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

  typedef long Period; // 100 nanoseconds

  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.

3.3.3 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 [51] 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 [50]) 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`.

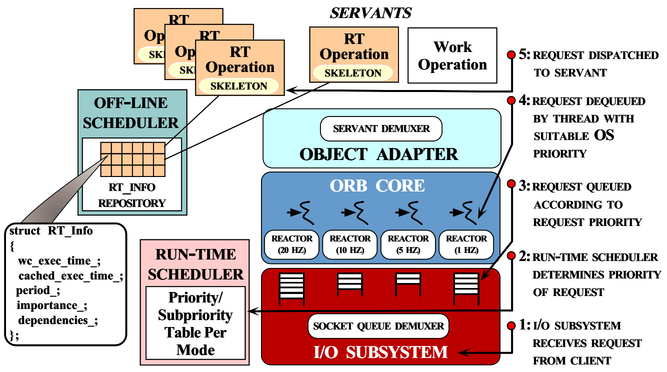
SERVANTS
RT Operation SKELETON
Work Operation
OFF-LINE SCHEDULER
RT_INFO REPOSITORY
struct RT_Info { wc_exec_time_; cached_exec_time_; period_; importance_; dependencies_; };
RUN-TIME SCHEDULER
Priority/ Subpriority Table Per Mode
SERVANT DEMUXER
OBJECT ADAPTER
ORB CORE
REACTOR (20 HZ)
REACTOR (10 HZ)
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

3.4 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* [52]. 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 dispatching*, 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 [62].

RT_Info: As described in Section 3.3, 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.

3.5 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.

3.5.1 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]:

```
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,
```

[7]The remainder of the RT_Scheduler module IDL description is shown in Section 3.3.2.

```
          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. There is also a `destroy` operation that the application calls when a program exits. This operation allows the scheduler to release its dynamically allocated resources.

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 3.3.3. This division of the Scheduling Service interface into application and privileged sections is shown in Figure 12.

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

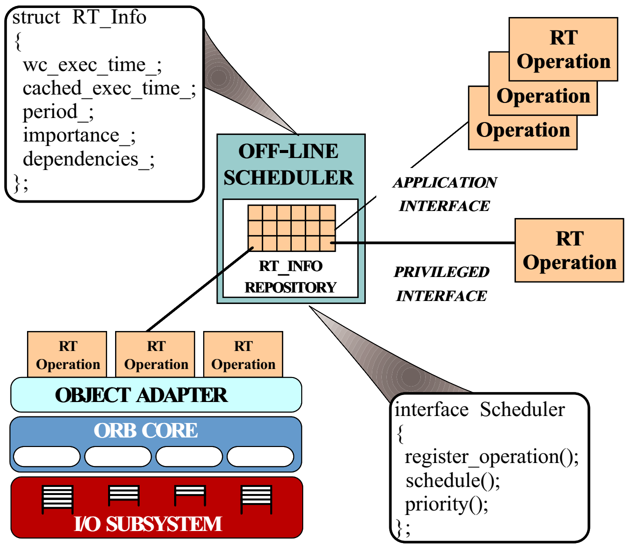

Figure 12: TAO's Two Scheduling Service Interfaces

priorities, when it is invoked, and what is stored in its internal database.

3.5.2 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.

3.5.3 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.

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_Infos` are not needed at run-time. Therefore, one space-saving optimization would be

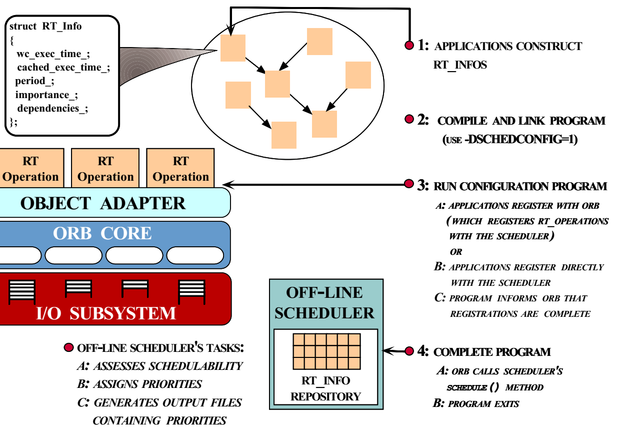
struct RT_Info
{
wc_exec_time_;
cached_exec_time_;
period_;
importance_;
dependencies_;
};
RT Operation
RT Operation
RT Operation
OBJECT ADAPTER
ORB CORE
I/O SUBSYSTEM
OFF-LINE SCHEDULER
RT_INFO REPOSITORY
1: APPLICATIONS CONSTRUCT RT_INFOS
2: COMPILE AND LINK PROGRAM (USE -DSCHEDCONFIG=1)
3: RUN CONFIGURATION 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

to conditionally compile `RT_Infos` 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.

3.5.4 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 3.5.2, 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.

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.

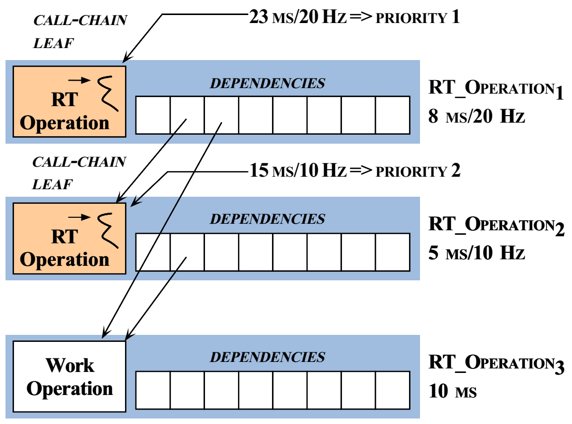

Figure 14: The RT_Info Repository

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.

3.5.5 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.*.

3.5.6 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.

4 Performance Experiments

Our past experience pinpointing performance bottlenecks in middleware like Web servers [63], and CORBA ORBs [21] demonstrates the efficacy of a measurement-driven research methodology. This section describes the results of an experiment that illustrates why conventional ORBs are unsuited for applications with real-time requirements. Future work will investigate the real-time performance of TAO in detail.

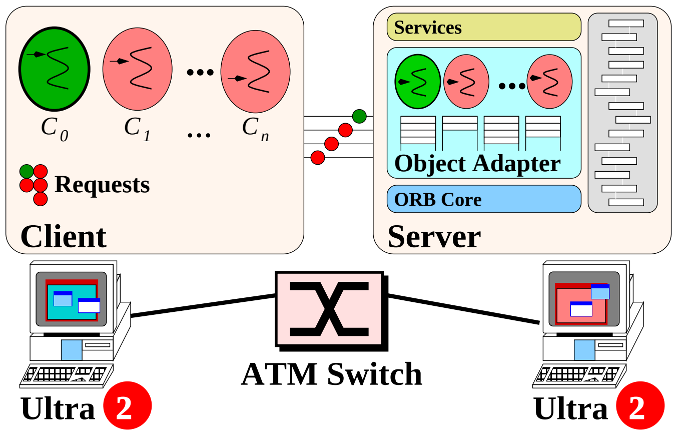

Figure 15: Testbed for ORB analysis

4.1 Measuring ORB Priority Inversion

This experiment measures the *degree of priority inversion* in an ORB. Priority inversion is the condition that occurs when the execution of a high priority thread is blocked by a lower priority thread. Priority inversion often occurs when threads running at different priorities share common I/O channels. It is hard to eliminate priority inversion completely; if it cannot be avoided, *bounded priority inversion* is desirable. This means that the amount of time a higher priority task is waiting due to a lower priority task must have a tight upper bound.

[14] describes one approach to control priority inversion, using *priority inheritance*. Priority inheritance temporarily increases the priority of a lower priority task when the system detects that a higher priority task cannot proceed due to dependencies on a lower priority task. However, this basic priority inheritance protocol can run into problems as well, such as formation of deadlock, and of chained blocking [64]. To address these problems, *priority ceiling* protocols can be employed. These protocols involve assigning a *ceiling* priority to the shared resource [64].

4.1.1 Experimental Setup

The experimental testbed is depicted in Figure 15. The experiments was conducted using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The LattisCell 10114 is a 16 Port, OC3 155 Mbs/port switch. Each UltraSPARC-2 contains 2 168 MHz CPUs with a 1 Megabyte cache per-CPU, 256 Mbytes of RAM, and an ENI-155s-MF ATM adaptor card that supports 155 Megabits per-sec (Mbps) SONET multi-mode 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 K). This allows up to eight switched virtual connections per card.

We selected two popular commercial multi-threaded CORBA implementations for the priority inversion tests: Iona's MT-Orbix v2.2, and Visigenic's Visibroker v2.0. The test used one high priority client C_0 and n low priority clients, $C_1 \ldots C_n$. The priority of the clients is determined as follows: each client makes time-constrained two-way CORBA operation invocations, *i.e.*, the client requires the operation invocation to complete within a predefined amount of time, referred to as the *deadline*. The high-priority client has an earlier deadline than the low-priority clients. Therefore, its operation invocations must complete earlier than the low-priority clients.

The server uses the thread-per-session concurrency model. In this model, a new thread is created to handle each client connection. The server supports the notion of the client priority using the Active Object pattern [52], as follows. Each client requests the creation of a servant object using the `create_servant` method provided by a Server Factory. The client indicates its priority as an argument to this method. The Factory creates a new *servant* object for each client. It also creates a new thread, called the *servant thread*, to handle all future requests from this client. The priority indicated by the client when it calls the `create_servant` method is used by the Factory to select an appropriate priority for the *servant thread*. For the high priority client, the server uses the *highest real-time priority* available with the Solaris operating system. For the low priority client, the server uses the *lowest real-time priority* available on Solaris.

As the number of low-priority clients increases, the number of low-priority requests on the server also increases. When the load becomes high, these low-priority clients begin to contend with high-priority requests made by C_0. ORBs that avoid priority inversion by implementing preemptive GIOP protocol processing can satisfy the deadline of the high priority client even in the presence of heavy low priority load. As shown in the results below, however, conventional ORBs allow an unlimited number of low priority clients to make CORBA requests since they do not perform *admission control*, as discussed in Section 1.2.

4.1.2 Results for MT-Orbix and VisiBroker

Figure 16 and Figure 17 plot the response times experienced by the high-priority client C_0, and the average response time experienced by the low-priority clients $C_1 \ldots C_n$, as we increase the value of n. These figures indicate that both Orbix and Visibroker exhibit extensive priority inversion. In particular, as the number of low priority clients increases, the latency observed by the high priority client increases rapidly. Since the server uses a higher real-time priority thread to handle high priority client requests, the latency seen by the higher priority clients should *not* be affected by the presence of lower priority requests.

The increase in the latency observed by the high priority client is due to priority inversion in various layers of the ORB endsystems, as described below:

• **OS I/O Subsystem:** The Solaris I/O subsystem does not perform *preemptible prioritized protocol processing, i.e.*, the protocol processing of lower priority packets is *not* deferred due to the arrival of a higher priority packet. Thus, incoming packets are processed according to their *order of arrival* rather than their *priority*. For instance, if a low priority request arrives before a high priority request, the I/O subsystem will process the lower priority packet *before* the higher priority packet. The amount of time spent in the low-priority servant represents the priority inversion. TAO addresses these problems using the Gigabit Real-time I/O Subsystem discussed in Section 2.1.

• **ORB Core:** The ORB Core implements the GIOP protocol. It thus sends/receives GIOP packets to/from the I/O subsystem and is responsible for processing these packets. Current GIOP mappings (such as IIOP) do not communicate request priority with each request. Therefore, the ORB Core is unaware of the priority of the request. Hence, the ORB Core for MT-Orbix and VisiBroker process GIOP packets in their order of arrival, which leads to priority inversion. TAO implements the RIOP protocol discussed in Section 2.2.1, which can include QoS information with each request. This information can be used to perform prioritized protocol processing, thus alleviating priority inversion in TAO's ORB Core.

• **Object Adapter:** The Object Adapter for MT-Orbix and VisiBroker do not perform prioritized demultiplexing of requests. In addition, these ORB implementations perform *layered demultiplexing*, which causes priority inversion and other performance penalties [20]. Section 2.3 describes the design of TAO's real-time Object Adapter and how it eliminates priority inversion.

These results illustrate the non-deterministic performance seen by applications running atop conventional ORBs that lack real-time features. In addition, the results show that priority inversion is a significant problem in these ORBs, and thus they are unsuitable for applications with deterministic real-time requirements.

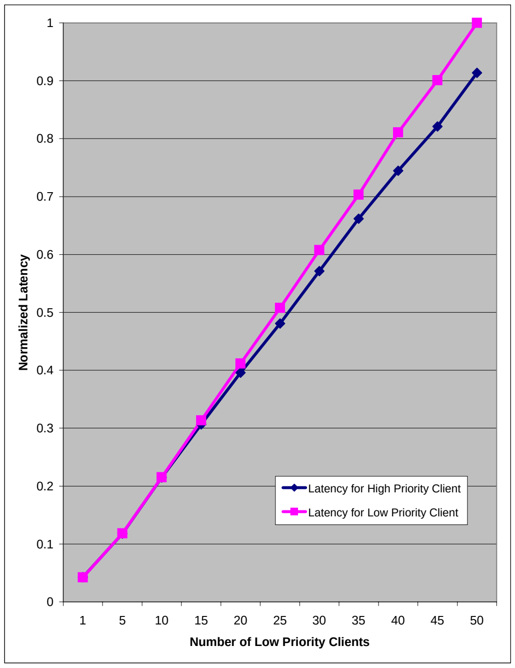

Figure 16: Priority Inversion in Orbix

4.1.3 Results for TAO's Real-time Event Channel

To illustrate how TAO addresses the priority inversion problems discussed above, we performed an experiment similar to the above with the TAO Real-time Event Channel [10]. As discussed in Section 3, the Event Channel currently implements several key features of TAO's Real-time Object Adapter (ROA), such as real-time dispatching of requests (events), and real-time scheduling of clients/servants (suppliers/consumers).

Similar to the experiments performed with Orbix and Visibroker, we created a high priority client and n low priority clients. Each client had its own servant object. Thus the CORBA clients used in the tests with Orbix and Visibroker were modeled as Event suppliers and CORBA servants were modeled as Event consumers. The Event Channel Scheduler assigns appropriate real-time priorities to the servants, similar to the Server Factory in the experiments performed with Orbix and Visibroker.

Our experiment measured the *latency* observed by the channel clients, *i.e.*, the time taken for the Event Channel to demultiplex events, dispatch them to the servant, and for the servant to handle these requests. Figure 18 depicts this latency observed by the high priority client as the number of low priority clients increase. These results illustrate that the latency observed by the high priority client is not significantly adversely affected as the number of low priority clients increases. Therefore, the Event Channel correctly handles the priorities of the clients and does not suffer from priority inversion. These results serve as a *proof of concept* of the prioritized request processing capabilities in TAO's real-time Object Adapter.

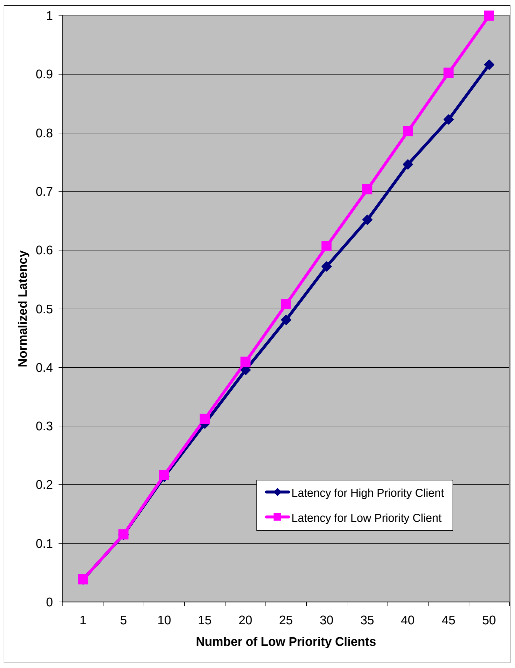

Figure 17: Priority Inversion in Visibroker

Real-time middleware is an emerging field of study. An increasing number of research efforts are focusing on integrating QoS and real-time scheduling into middleware like CORBA. This section compares our work on TAO with related QoS middleware integration research.

CORBA-related QoS research: Krupp, *et al.*, [65] at MITRE Corporation were among the first to elucidate the requirements of real-time CORBA systems. A system consisting of a commercial off-the-shelf RTOS, a CORBA-compliant ORB, and a real-time object-oriented database management system is under development [66]. Similar to the initial approach provided by TAO, their initial static scheduling approach uses RMS, though a strategy for dynamic deadline monotonic scheduling support has been designed [67].

Wolfe, *et al.*, are developing a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) [68]. The system supports expression and enforcement of dynamic end-

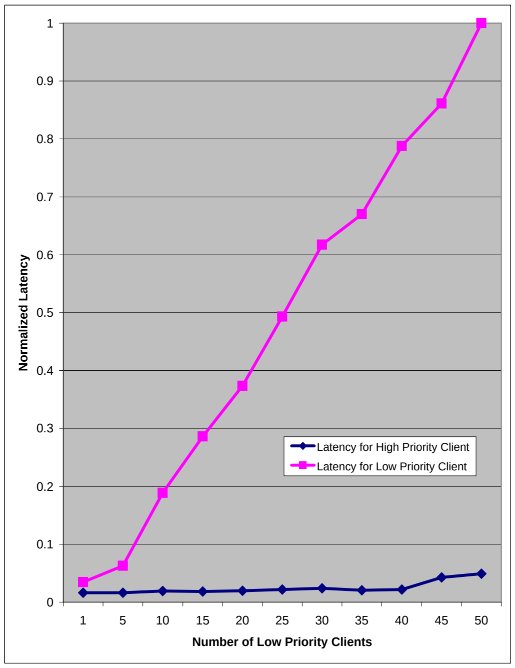

Figure 18: TAO Event Channel Performance

to-end timing constraints through timed distributed operation invocations (`TDMI`s) [15]. A `TDMI` corresponds to TAO's `RT_Operation` [23]. Likewise, an `RT_Environment` structure contains QoS parameters similar to those in TAO's `RT_Info`.

One difference between TAO and the URI approaches is that `TDMI`s express required timing constraints, *e.g.*, deadlines relative to the current time, whereas `RT_Operation`s publish their resource, *e.g.*, CPU time, requirements. The difference in approaches may reflect the different time scales, seconds versus milliseconds, respectively, and scheduling requirements, dynamic versus static, of the initial application targets. However, the approaches should be equivalent with respect to system schedulability and analysis.

In addition, NRaD/URI supply a new CORBA Global Priority Service (analogous to TAO's Scheduling Service), and augment the CORBA Concurrency and Event Services. The initial implementation uses *EDF within importance level* dynamic, on-line scheduling, supported by global priorities. A global priority is associated with each `TDMI`, and all processing associated with the TDMI inherits that priority. In contrast, TAO's initial Scheduling Service was static and off-line; it uses importance as a "tie-breaker" following the analysis of other requirements such as data dependencies. Both NRaD/URI and TAO readily support changing the scheduling policy by encapsulating it in their CORBA Global Priority and Scheduling Services, respectively.

The QuO project at BBN [40] has defined a model for communicating changes in QoS characteristics between applications, middleware, and the underlying endsystems and network. The QuO model uses the concept of a *connection* between a client and an object to define QoS characteristics. These characteristics are treated as first-class objects. Objects can be aggregated to enable characteristics to be defined at various levels of granularity, *e.g.*, for a single method invocation, for all method invocations on a group of objects, and similar combinations. The QuO model also uses several QoS definition languages (QDLs) that describe the QoS characteristics of various objects, such as expected usage patterns, structural details of objects, and resource availability.

The QuO architecture differs from our work on real-time QoS provisioning in TAO since QuO does not provide hard real-time guarantees of ORB endsystem CPU scheduling. Furthermore, the QuO programming model involves the use of several QDL specifications, in addition to OMG IDL, based on the separation of concerns advocated by Aspect-Oriented Programming (AoP) [69]. We believe that although the AOP paradigm is powerful, the proliferation of definition languages may be overly complex for common application use-cases. Therefore, the TAO programming model focuses on the `RT_Operation` and `RT_Info` QoS specifiers, which can be expressed in standard OMG IDL and integrated seamlessly with the existing CORBA programming model.

The Realize project at UCSB [39] supports soft real-time resource management of CORBA distributed systems. Realize aims to reduce the difficulty of developing real-time systems and to permit distributed real-time programs to be programmed, tested, and debugged as easily as single sequential programs. The key innovations in Realize are its integration of distributed real-time scheduling with fault-tolerance, of fault-tolerance with totally-ordered multicasting, and of totally-ordered multicasting with distributed real-time scheduling, within the context of object-oriented programming and existing standard operating systems. Realize can be hosted on top of TAO [39].

The Epiq project [70] defines an open real-time CORBA scheme that provides QoS guarantees and run-time scheduling flexibility. Epiq explicitly extends TAO's off-line scheduling model to provide on-line scheduling. In addition, Epiq allows clients to be added and removed dynamically via an admission test at run-time. The Epiq project is work-in-progress and empirical results are not yet available.

Non-CORBA-related QoS research: The ARMADA project [71, 72] defines a set of communication and middleware services that support fault-tolerant and end-to-end guarantees for real-time distributed applications. ARMADA provides real-time communication services based on the X-kernel and the Open Group's MK microkernel. This infrastructure provides a foundation for constructing higher-level real-time middleware services.

TAO differs from ARMADA in that most of the real-time infrastructure features in TAO are integrated into its ORB Core. In addition, TAO implements the OMG's CORBA standard, while also providing the hooks that are necessary to integrate with an underlying real-time I/O subsystem and OS. Thus, the real-time services provided by ARMADA's communication system can be utilized by TAO's ORB Core to support a vertically and horizontally integrated real-time system.

Rajkumar, *et al.*, [73] at the Carnegie Mellon University Software Engineering Institute, developed a real-time Publisher/Subscriber model. It is functionally similar to the TAO's Real-time Event Service [10]. For instance, it uses real-time threads to prevent priority inversion within the communication framework.

The CMU model does not utilize any QoS specifications from publishers (event suppliers) or subscribers (event consumers). Therefore, scheduling is based on the assignment of request priorities, which is not addressed by the CMU model. In contrast, TAO's Scheduling Service and real-time Event Service utilize QoS parameters from suppliers and consumers to assure resource access via priorities. One interesting aspect of the CMU Publisher/Subscriber model is the separation of priorities for subscription and data transfer. By handling these activities with different threads, with possibly different priorities, the impact of on-line scheduling on real-time processing can be minimized.

5 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 **??**.

TAO is currently being deployed at Boeing in St. Louis, MO, where it is being used to develop operation flight programs for next-generation avionics systems. Source code for the TAO ORB is available at `www.cs.wustl.edu/∼schmidt/TAO.html`.

Acknowledgements

We would like to thank Frank Buschmann for his extensive comments on this paper and Tim Harrison, Rajeev Bector, and Irfan Pyarali for help with earlier drafts of this paper. In addition, we gratefully acknowledge the support and direction of the Boeing Principal Investigator, Bryan Doerr.

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, *CORBA Messaging 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. L. 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,” *IEEE Communications Magazine*, to appear 1999.

[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] Object Management Group, *Realtime CORBA 1.0 Joint Submission*, OMG Document orbos/98-12-05 ed., December 1998.

[45] 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 4^{th} IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

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

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

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

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-time Synchronization," *IEEE Transactions on Computers*, vol. 39, September 1990.

[65] B. Thuraisingham, P. Krupp, A. Schafer, and V. Wolfe, "On Real-Time Extensions to the Common Object Request Broker

Architecture," in *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Experiences with CORBA*, ACM, Oct. 1994.

[66] "Statement of Work for the Extend Sentry Program, CPFF Project, ECSP Replacement Phase II," Feb. 1997. Submitted to OMG in response to RFI ORBOS/96-09-02.

[67] G. Cooper, L. C. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thuraisingham, S. Wohlever, and V. F. Wolfe, "Real-Time CORBA Development at MITRE, NRaD, Tri-Pacific and URI," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

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

[69] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[70] W. Feng, U. Syyid, and J.-S. Liu, "Providing for an Open, Real-Time CORBA," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[71] A. Mehra, A. Indiresan, and K. G. Shin, "Structuring Communication Software for Quality-of-Service Guarantees," *IEEE Transactions on Software Engineering*, vol. 23, pp. 616–634, Oct. 1997.

[72] T. Abdelzaher, S. Dawson, W.-C.Feng, F.Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, and H. Zou, "ARMADA Middleware Suite," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[73] R. Rajkumar, M. Gagliardi, and L. Sha, "The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation," in *First IEEE Real-Time Technology and Applications Symposium*, May 1995.

[74] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "Implementing Real-Time Robotic Systems using CHIMERA II," in *Proceedings of 1990 IEEE International Conference on Robotics and Automation*, (Cincinnatti, OH), 1992.

[75] A. Gokhale and D. C. Schmidt, "Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems," in *Proceedings of INFOCOM '99*, Mar. 1999.

[76] A. Gokhale, I. Pyarali, C. O'Ryan, D. C. Schmidt, V. Kachroo, A. Arulanthu, and N. Wang, "Design Considerations and Performance Optimizations for Real-time ORBs," in *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[77] C. D. Gill, D. L. Levine, and D. C. Schmidt, "Evaluating Strategies for Real-Time CORBA Dynamic Scheduling," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1999, to appear.

[78] 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.

[79] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[80] K. Ramamritham, C. Shen, O. Gonzáles, S. Sen, and S. Shirgurkar, "Using Windows NT for Real-time Applications: Experimental Observations and Recommendations," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

A Adapting Scheduling Strategies to Support TAO

TAO's real-time Scheduling Service uses a strategized scheduler, which can implement any of several scheduling strategies including *Rate Monotonic Scheduling* (RMS) [36, 37], *Earliest Deadline First* (EDF) [36], *Minimum Laxity First* (MLF), and *Maximum Urgency First* (MUF) [50]. Each of these strategies carries certain benefits and limitations, which the body of this section addresses.

A.1 Rate Monotonic Analysis and Scheduling

In RMS, higher priorities are assigned statically to threads with faster rates. These priorities are fixed (*i.e.*, they are not modified by the operating system) and static (*i.e.*, they are not changed by the application). The operating system must support preemption, such that the highest priority, runnable thread(s) are always running on available processor(s).

Rate monotonic analysis (RMA), is used to validate RMS schedules. A great deal of research on RMA has been done at the Software Engineering Institute of Carnegie Mellon University. RMA consists of analytical methods and algorithms for evaluating timing behavior. Note that systems need not be scheduled with RMS to be analyzed using RMA; therefore, by selecting RMA TAO is not limited to RMS.

RMA provides a test of whether *schedulability* can or cannot be guaranteed for a system. In the context of deterministic real-time systems, schedulability is the ability of all operations in the system to meet their deadlines. We selected RMS as the first policy for TAO's Scheduling Service because it provides an optimum schedule, is straightforward to apply, and is relatively simple to analyze.

The TAO instantiation of RMA starts with the most strict rules and progressively relaxes restrictions until it reaches a form that can be used. The following restrictions allow the application of RMA [36, 37]:

- *All operations are independent*;

- *All operations are periodic*;
- *There are no interrupts*;
- *Context switching time is negligible*;
- *There is a single operation priority level*;
- *There is a single CPU*;
- *Operation periods are related harmonically*; and
- *All operation deadlines are at the ends of periods*.

Given the above restrictions, and knowledge of the computation time, C_t, and period, P_t, of each operation t, then the *schedulability test* is simply a comparison of the sum of the utilizations, $\sum_{t=1}^{n} \frac{C_t}{P_t}$, over each of the n operations in the program with 1. If the sum of the utilizations is less than or equal to 1, the operation set is schedulable; otherwise, it is not.

Many of these restrictions can be relaxed for TAO in deterministic real-time environments, as follows:

Interdependent operations: When operations are not independent, scheduling analysis must (conservatively) consider the time that a thread may be blocked by one of lower priority due to synchronization. With sufficient analysis of system participants and their behaviors, this blocking can be eliminated by explicit specification of dependencies and resultant execution ordering. In practice, however, such explicit dependency specification only may be feasible for deterministic real-time systems that can be analyzed statically. In such systems, thread activity can effectively be determined prior to run-time. Our `RT_Info` IDL `struct` supports this type of off-line analysis.

In statistical real-time systems that have dynamically changing resource requirements, operation interdependencies are harder to analyze. For instance, there is a potential for *priority inversion* if threads of different priorities can synchronize. To achieve optimum resource utilization, it is best to prevent these situations. However, if they can occur, the analysis must (conservatively) consider the time that a thread may be blocked by a lower priority due to synchronization.

Aperiodic operations: Aperiodic operations can be modeled as periodic operations, assuming the worst (fastest) possible rate that the operations can execute.

Interrupts: Interrupts can be handled in the analysis given their execution times and maximum possible rate. The usual drawback, however, is that the analysis is conservative. It assumes that the interrupts will occur at that maximum possible rate; while necessary, this assumed rate is usually not realized in practice. The result is reduced effective CPU utilization because the CPU must be "reserved" for interrupts that may not always occur.

Context switching: Context switching time can be accounted by charging each switch to the execution time of the thread that is swapped out.

Multiple operation priority levels: TAO's real-time Scheduling Service generates operation priorities as its output. It assigns an OS-specific priority to each thread in the application, *e.g.*, using RMS. Each operation in the thread is then assigned the priority of that thread. For operations in more than one thread, the highest priority is assigned. RMA can be applied when there are multiple priority levels if there is preemption, which is supported by TAO's Object Adapter. If preemption is not immediate, then it must be accounted for in the analysis; an example is the analysis of RTUs [28].

Multiple CPUs: Currently, our RMA analysis assumes TAO's Object Adapter dispatches client requests on a single CPU. Therefore, all work can be scheduled on that CPU in isolation. The first step towards scheduling on multiple CPUs will be to allocate threads manually to the separate CPUs and to schedule each CPU separately, considering interprocessor communication as interrupts. Further refinement of the analysis will take the actual priority of interprocessor events into account.

Operation periods are related harmonically: If operation periods are not related harmonically, then the *utilization bound* (*i.e.*, the maximum utilization below which the operation set is guaranteed to be schedulable) is $n \times (2^{1/n} - 1)$, where n is the number of operations in the set. This function approaches 0.693 as n grows large. However, if all of the operation periods are related harmonically (*e.g.*, 30 Hz, 15 Hz, 5 Hz, etc.), the utilization bound is 1. Intuitively, this is because the operation periods "fit" into the largest operation period. For applications that can have harmonically related operation periods, it is clearly advantageous to use these harmonic relations to maximize CPU utilization.

All operation deadlines are at the ends of periods: Preperiod operation deadlines can be modeled by adjusting the utilization bound.

A.1.1 Purely Dynamic Scheduling Strategies

This section reviews two well known purely dynamic scheduling strategies, Earliest Deadline First (EDF) [36, 37], and Minimum Laxity First (MLF) [50]. These strategies are illustrated in Figure 19 and discussed below. In addition, Figure 19 depicts the hybrid static/dynamic Maximum Urgency First (MUF) [50] scheduling strategy discussed in Section A.1.2.

Earliest Deadline First (EDF): EDF [36, 37] is a dynamic scheduling strategy that orders dispatches[8] of operations based on time-to-deadline, as shown in Figure 19. Operation executions with closer deadlines are dispatched before those with

[8]A *dispatch* is a particular execution of an *operation*.

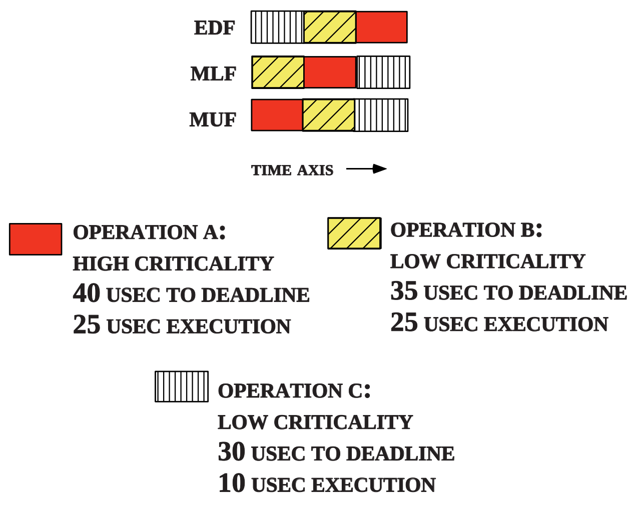

Figure 19: Dynamic Scheduling Strategies

more distant deadlines. The EDF scheduling strategy is invoked whenever a dispatch of an operation is requested. The new dispatch may or may not preempt the currently executing operation, depending on the mapping of priority components into thread priorities discussed in Section B.5.5.

A key limitation of EDF is that an operation with the earliest deadline is dispatched whether or not there is sufficient time remaining to complete its execution prior to the deadline. Therefore, the fact that an operation cannot meet its deadline will not be detected until *after* the deadline has passed.

If the operation is dispatched even though it cannot complete its execution prior to the deadline, the operation consumes CPU time that could otherwise be allocated to other operations. If the result of the operation is only useful to the application prior to the deadline, then the entire time consumed by the operation is essentially wasted.

Minimum Laxity First (MLF): MLF [50] refines the EDF strategy by taking into account operation execution time. It dispatches the operation whose *laxity* is least, as shown in Figure 19. Laxity is defined as the time-to-deadline minus the remaining execution time.

Using MLF, it is possible to detect that an operation will not meet its deadline *prior* to the deadline itself. If this occurs, a scheduler can reevaluate the operation before allocating the CPU for the remaining computation time. For example, one strategy is to simply drop the operation whose laxity is not sufficient to meet its deadline. This strategy may decrease the chance that subsequent operations will miss their deadlines, especially if the system is overloaded transiently.

Evaluation of EDF and MLF:

- **Advantages:** From a scheduling perspective, the main advantage of EDF and MLF is that they overcome the utilization limitations of RMS. In particular, the utilization phasing penalty described in Section **??** that can occur in RMS is not a factor since EDF and MLF prioritize operations according to their dynamic run-time characteristics.

  EDF and MLF also handle harmonic and non-harmonic periods comparably. Moreover, they respond flexibly to invocation-to-invocation variations in resource requirements, allowing CPU time unused by one operation to be reallocated to other operations. Thus, they can produce schedules that are optimal in terms of CPU utilization [36]. In addition, both EDF and MLF can dispatch operations within a single static priority level and need not prioritize operations by rate [36, 50].

- **Disadvantages:** From a performance perspective, one disadvantage to purely dynamic scheduling approaches like MLF and EDF is that their scheduling strategies require higher overhead to evaluate at run-time. In addition, these purely dynamic scheduling strategies offer no control over *which* operations will miss their deadlines if the schedulable bound is exceeded. As operations are added to the schedule to achieve higher utilization, the margin of safety for *all* operations decreases. Therefore, the risk of missing a deadline increases for every operation as the system become overloaded.

A.1.2 Maximum Urgency First

The Maximum Urgency First (MUF) [50] scheduling strategy supports both the deterministic rigor of the static RMS scheduling approach and the flexibility of dynamic scheduling approaches such as EDF and MLF. MUF is the default scheduler for the Chimera real-time operating system (RTOS) [74]. TAO supports a variant of MUF in its strategized CORBA scheduling service framework, which is discussed in Section B.

MUF can assign both static *and* dynamic priority components. In contrast, RMS assigns all priority components statically and EDF/MLF assign all priority components dynamically. The hybrid priority assignment in MUF overcomes the drawbacks of the individual scheduling strategies by combining techniques from each, as described below:

Criticality: In MUF, operations with higher *criticality* are assigned to higher static priority levels. Assigning static priorities according to criticality prevents operations critical to the application from being preempted by non-critical operations.

Ordering operations by application-defined criticality reflects a subtle and fundamental shift in the notion of priority assignment. In particular, RMS, EDF, and MLF exhibit a rigid mapping from empirical operation characteristics to a single priority value. Moreover, they offer little or no control over which operations will miss their deadlines under overload conditions.

In contrast, MUF gives applications the ability to distinguish operations arbitrarily. MUF allows control over *which* operations will miss their deadlines. Therefore, it can protect a critical *subset* of the entire set of operations. This fundamental shift in the notion of priority assignment leads to the generalization of scheduling and analysis techniques discussed in Section B and Appendix **??**.

Dynamic Subpriority: An operation's dynamic subpriority is evaluated whenever it must be compared to another operation's dynamic subpriority. For example, an operation's dynamic subpriority is evaluated whenever it is enqueued in or dequeued from a dynamically ordered dispatching queue. At the instant of evaluation, dynamic subpriority in MUF is a function of the the laxity of an operation.

An example of such a simple dynamic subpriority function is the inverse of the operation's laxity.[9] Operations with the smallest positive laxities have the highest dynamic subpriorities, followed by operations with higher positive laxities, followed by operations with the most negative laxities, followed by operations with negative laxities closer to zero. Assigning dynamic subpriority in this way provides a consistent ordering of operations as they move through the *pending* and *late* dispatching queues, as described below.

By assigning dynamic subpriorities according to laxity, MUF offers higher utilization of the CPU than the statically scheduled strategies. MUF also allows deadline failures to be detected *before* they actually occur, except when an operation that would otherwise meet its deadline is preempted by a higher criticality operation. Moreover, MUF can apply various types of error handling policies when deadlines are missed [50]. For example, if an operation has negative laxity prior to being dispatched, it can be demoted in the priority queue, allowing operations that can still meet their deadlines to be dispatched instead.

Static Subpriority: In MUF, *static subpriority* is a static, application-specific, optional priority. It is used to order the dispatches of operations that have the same criticality and the same dynamic subpriority. Thus, static subpriority has lower precedence than either criticality or dynamic subpriority.

Assigning a unique static subpriority to operation that have the same criticality ensures a total dispatching ordering of operations at run-time, for any operation laxity values having the same criticality. A total dispatching ordering ensures that for a given arrival pattern of operation requests, the dispatching order will always be the same. This, in turn, helps improve the reliability and testability of the system.

[9]To avoid division-by-zero errors, any operation whose laxity is in the range $\pm\epsilon$ can be assigned (negative) dynamic subpriority $-1/\epsilon$ where ϵ is the smallest positive floating point number that is distinguishable from zero. Thus, when the laxity of an operation reaches ϵ, it is considered to have missed its deadline.

The variant of MUF used in TAO's strategized scheduling service enforces a complete dispatching ordering by providing an `importance` field in the TAO `RT_Info` CORBA operation QoS descriptor [23], which is shown in Section **??**. TAO's scheduling service uses `importance`, as well as a topological ordering of operations, to assign a unique static subpriority for each operation within a given criticality level.

Incidentally, the original definition of MUF in [50] uses the terms *dynamic priority* and *user priority*, whereas we use the term *dynamic subpriority* and *static subpriority* for TAO's scheduling service. We selected different terminology to indicate the subordination to static priority. These terms are interchangeable when referring to MUF, however.

It is not strictly necessary to know all operations in advance in order to schedule them using the canonical definitions of EDF or MLF. However, the real-time applications we have worked with do exhibit this useful property. If all operations are known in advance, off-line analysis of schedule feasibility is possible for RMS, EDF, MLF, and MUF.

The output of each of the scheduling strategies in TAO is a *schedule*. This schedule defines a set of operation dispatching priorities, dispatching subpriorities, and a minimum critical dispatching priority. Our goal in this appendix is to present a feasibility analysis technique for these schedules, that is independent of the specific strategy used to produce a particular schedule. Such an analysis technique must establish invariants that hold across all urgency and dispatching priority mappings. By doing this, the off-line schedule feasibility analysis (1) decouples the application from the details of a particular scheduling strategy, and (2) allows alternative strategies to be compared for a given application .

The remainder of this appendix is organized as follows. Section A.2 discusses the notion of a schedule's *frame size*. Section A.3 describes how we measure a schedule's CPU utilization. Finally, Section A.4 describes the generalized schedule feasibility analysis technique, which is based on a schedule's utilization, frame size, and the respective priorities of the operations.

A.2 Frame Size

The frame size for a schedule is the minimum time that can contain all possible phasing relationships between all operations. The frame size provides an invariant for the largest time within which all operation executions will fit. This assumes, of course, that the scheduling parameters, such as rates and worst-case execution times, specified by applications are not exceeded by operations at run-time.

When the periods of all operations are integral multiples of one another, *e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz, the operations are said to be *harmonically related*. Harmonically related operations have completely nested phasing relationships. Thus, the arrival pattern of each subsequently shorter period fits exactly within the next longer period. For harmonically related operations, the frame size is simply the longest operation period.

Operations that are not harmonically related come into and out of phase with one another. Therefore, they do not exhibit the nesting property. Instead, the pattern of arrivals only repeats after all periods come back into the same phasing relationships they had at the beginning.

This observation leads to the invariant that covers both the harmonic and non-harmonic cases. The frame size in both cases is the product of all non-duplicated factors of all operation periods. For non-harmonic cases, we calculate this value by starting with a frame size of one time unit and iterating through the set of unique operation periods. For each unique period, we (possibly) expand the frame size by multiplying the previous frame size by the greatest common divisor of the previous frame size and the operation period. For harmonic cases, all operation periods are factors of the longest operation period. Therefore, the longest operation period is the frame size.

Figure 20 depicts the relationships between operation periods and frame size for both the harmonic and non-harmonic cases. For harmonically related operation rates, all of the

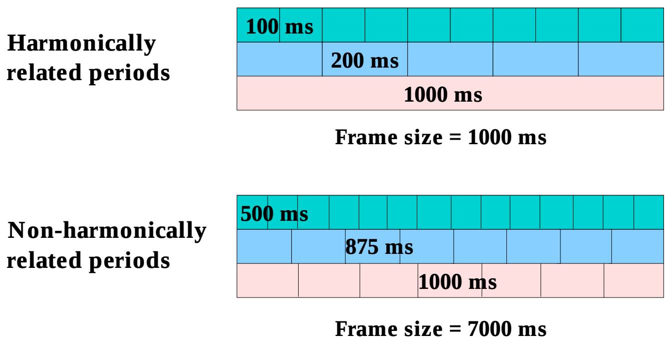

Figure 20: Frame Size Examples for Harmonic and Non-Harmonic Cases

smaller periods fit evenly into the largest period. Therefore, the largest operation period *is* the frame size. For non-harmonically related rates, the frame size is larger than the largest operation period, because it is a multiple of all of the operation periods.

A.3 Utilization

Total CPU utilization is the sum of the actual execution times used by all operation dispatches over the schedule frame size, divided by the frame size itself. TAO's strategized scheduling service calculates the maximum total utilization for a given schedule by summing, over all operations, the fraction of each operation's period that is consumed by its worst-case execution time, according to the following formula:

$$\mathcal{U} = \sum_{\forall k} C_k / T_k$$

where, for each operation k, C_k is its worst case execution time, and T_k is its period.

In addition to total utilization, TAO's scheduling service calculates the CPU utilization by the set of critical operations. This indicates the percentage of time the CPU is allocated to operations whose completion prior to deadline is to be enforced. Operations whose assigned dispatching priority is greater than or equal to the minimum critical priority bound are considered to be in the critical set. In the RMS, EDF, and MLF scheduling strategies, the entire schedule is considered critical, so the critical set utilization is the same as total utilization.

If the total utilization exceeds the *schedulable bound*, TAO's scheduling service also stores the priority level previous to the one that exceeded the schedulable bound. This previous priority level is called the *minimum guaranteed priority level*. Operations having dispatching priority greater than or equal to the minimum guaranteed priority level are assured of meeting their deadlines. In contrast, operations having dispatching priority immediately below the minimum guaranteed priority level may execute prior to their deadlines, but are not assured of doing so. If the total utilization does not exceed the schedulable bound, the lowest priority level in the system is the minimum guaranteed priority level, and all operations are assured of meeting their deadlines.

A.4 Schedule Feasibility

It may or may not be possible to achieve a *feasible* schedule that utilizes 100% of the CPU. Achieving 100% utilization depends on the phasing relationships between operations in the schedule, and the scheduling strategy itself. The maximum percentage of the CPU that can be utilized is called the *schedulable bound*.

The schedulable bound is a function of the scheduling strategy and in some cases of the schedule itself. A schedule is *feasible* if and only if all operations in the critical set are assured of meeting their deadlines. The critical set is identified by the minimum critical priority. All operations having dispatching priority greater than or equal to the minimum critical priority are in the critical set.

The schedulability of each operation in the critical set depends on the worst-case operation arrival pattern, which is called the *critical instant*. The critical instant for an operation occurs when the delay between its arrival and its completion is maximal [36]. For the preemptive-by-urgency dispatching model described in Section B.5.6, the critical instant for an operation occurs when it arrives simultaneously with all other operations.

For other dispatching models, the critical instant for a given operation differs slightly. It occurs only when the operation arrives immediately after another operation that will cause it the greatest *additional* preemption delay was dispatched. Further, it only occurs when the operation arrives simultaneously with all operations other than the one causing it additional preemption delay. If an operation is schedulable at its critical instant, it is assured of schedulability under any other arrival pattern of the same operations.

A key research challenge in assessing schedule feasibility is determining whether each operation has sufficient time to complete its execution prior to deadline. The deadline for an operation at its critical instant falls exactly at the critical instant plus its period. Not only must a given operation be able to complete execution in that period, it must do so in the time that is not used by preferentially dispatched operations. All operations that have higher dispatching priority than the current operation will be dispatched preferentially. All operations that have the same dispatching priority, but have deadlines at or prior to the deadline of the current operation, must also be considered to be dispatched preferentially.

The goal of assessing schedule feasibility off-line in a way that (1) is independent of a particular strategy, and (2) correctly determines whether each operation will meet its deadline, motivates the following analysis. TAO's strategized scheduling service performs this analysis for each operation off-line. We call the operation upon which the analysis is being performed the *current operation*. The number of arrivals, during the period of the current operation, of an operation having higher dispatching priority than the current operation is given by $\lceil T_c/T_h \rceil$, where T_c and T_h are the respective periods of the current operation and the higher priority operation. The time consumed by the higher priority operation during the period of the current operation is given by $\lfloor T_c/T_h \rfloor C_h + \min\,(T_c - \lfloor T_c/T_h \rfloor T_h, C_h)$, where the min function returns the minimum of the values, and C_h is the computation time used for each dispatch of the higher priority operation.

Similarly, the number of deadlines of another operation having the same dispatching priority as the current operation is given by $\lfloor T_c/T_s \rfloor$, where T_s is the period of the other operation having the same dispatching priority as the current operation. The time consumed by the other same priority operation over the period of the current operation is given by $\lfloor T_c/T_s \rfloor C_s$, where C_s is the computation time used by the other same priority operation [36]. Figure 21 illustrates the various possible relationships between the periods of operations in two priority levels.

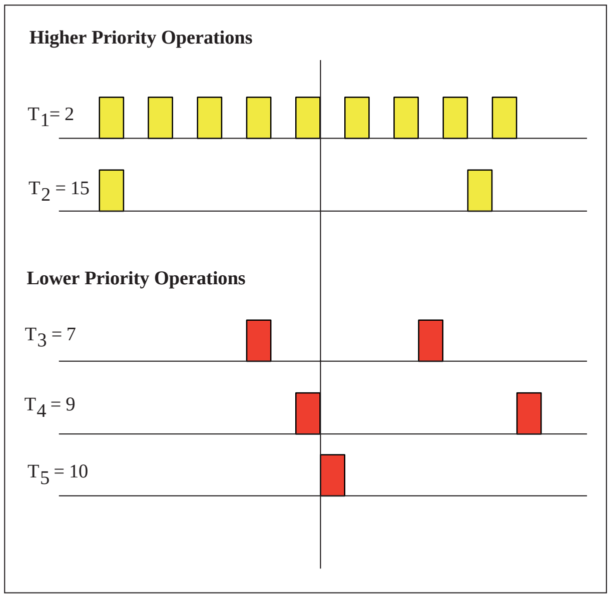

Figure 21: Schedulability of the Current Operation

Choosing the fourth operation, with period T_4, as the current operation, the number of arrivals of each of the higher priority operations is as expected: $\lceil T_4/T_1 \rceil = \lceil 9/2 \rceil = \lceil 4.5 \rceil = 5$ and $\lceil T_4/T_2 \rceil = \lceil 9/15 \rceil = \lceil 0.6 \rceil = 1$. The number of deadlines of operations having the same priority level is also as expected: $\lfloor T_4/T_3 \rfloor = \lfloor 9/7 \rfloor \doteq \lfloor 1.3 \rfloor = 1$ and $\lfloor T_4/T_4 \rfloor = \lfloor 9/9 \rfloor = \lfloor 1.0 \rfloor = 1$ and $\lfloor T_4/T_5 \rfloor = \lfloor 9/10 \rfloor = \lfloor 0.9 \rfloor = 0$.

Having established the time consumed by an operation having higher dispatching priority than the current operation as $\lfloor T_c/T_h \rfloor C_h + \min\,(T_c - \lfloor T_c/T_h \rfloor T_h, C_h)$, and the time consumed by an operation having the same dispatching priority as the current operation as $\lfloor T_c/T_s \rfloor C_s$, it is now possible to state the invariant that must hold for all operations having dispatching priority λ to be schedulable:

$$\forall \{ j, k \in \mathcal{S} \mid (p(j) = \lambda) \wedge (p(k) >= \lambda) \}$$

$$\left(\left[\begin{array}{l} C_{wcpd(j)} \; + \sum\limits_{p(k)>=\lambda} \lfloor T_j/T_k \rfloor C_k \; + \\ \\ \sum\limits_{p(k)>\lambda} \min\,(T_j - \lfloor T_j/T_k \rfloor T_k \;,\; C_k) \end{array} \right] <= T_j \right)$$

$\mathcal{S}$ is the set of all operations in the schedule. The function $p(j)$ simply returns the priority assigned to operation j. $C_{wcpd(j)}$ is the worst-case preemption delay for operation j. Operation j suffers a preemption delay if and only if it arrives while an operation in the same dispatching priority level that does not have a deadline within operation j's period is executing. Operations that have deadlines within operation j's period must be counted anyway, and thus do not impose any *additional* delay, should operation j arrive while they are executing. The worst-case preemption delay for operation j is the longest execution time of any operation that has a longer period: if there are no such operations, $C_{wcpd(j)}$ is zero.

For each current operation having dispatching priority λ to be schedulable, the following must hold. All deadlines of operations having the same dispatching priority or higher, including the deadline of the current operation itself, plus $C_{wcpd(j)}$, plus any time scheduled for higher priority operations that arrive within but do not have a deadline within the period of the current operations, must be schedulable within the period of the current operation. This invariant is evaluated for each decreasing dispatching priority level of a schedule, from the highest to the lowest. The lowest dispatching priority level for which the invariant holds is thus identified as the minimum priority for which schedulability of all operations can be guaranteed, known as the *minimum guaranteed priority*.

In summary, the schedule feasibility analysis technique presented in this appendix establishes and uses invariants that hold across all urgency and dispatching priority mappings. This gives applications the ability to examine different scheduling strategies off-line, and discard those that do not produce feasible schedules for their particular operation characteristics. Further, it decouples applications from the details of any particular scheduling strategy, so that changes in strategies to not require changes in their operation characteristics.

B The Design of TAO's Strategized Scheduling Service

TAO's scheduling service provides real-time CORBA applications with the flexibility to specify and use different scheduling strategies, according to their specific QoS requirements and available OS features. This flexibility allows CORBA applications to extend the set of available scheduling strategies *without* impacting strategies used by other applications. Moreover, it shields application developers from unnecessary details of their scheduling strategies. In addition, TAO's scheduling service provides a common framework to compare existing scheduling strategies and to empirically evaluate new strategies.

This section outlines the design goals and architecture of TAO's strategized scheduling service framework. After briefly describing TAO in Section B.1, Section B.2 discusses the design goals of TAO's strategized scheduling service. Section B.3 offers an overview of its architecture and operation. Section B.4 describes the design forces that motivate TAO's flexible Scheduling Service architecture. Finally, Section B.5 discusses the resulting architecture in detail.

B.1 Overview of TAO

TAO is a high-performance, real-time ORB endsystem targeted for applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements. The TAO ORB endsystem contains the network interface, OS, communication protocol, and CORBA-compliant middleware components and features shown in Figure 22. TAO supports the

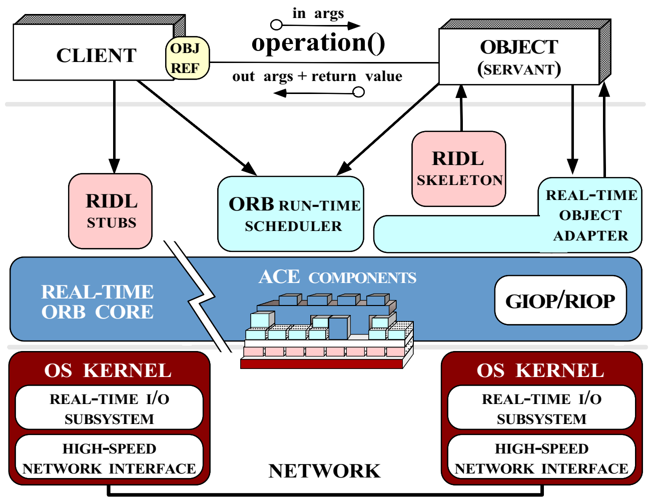

Figure 22: Components in the TAO Real-time ORB Endsystem

standard OMG CORBA reference model [1], with the following enhancements designed to overcome the shortcomings of conventional ORBs [45] for high-performance and real-time applications:

Real-time IDL Stubs and Skeletons: TAO's IDL stubs and skeletons efficiently marshal and demarshal operation parameters, respectively [75]. In addition, TAO's Real-time IDL (RIDL) stubs and skeletons extend the OMG IDL specifications to ensure that application timing requirements are specified and enforced end-to-end [68].

Real-time Object Adapter: An Object Adapter associates servants with the ORB and demultiplexes incoming requests to servants. TAO's real-time Object Adapter [76] uses perfect hashing [56] and active demultiplexing [20] optimizations to dispatch servant operations in constant $O(1)$ time, regardless of the number of active connections, servants, and operations defined in IDL interfaces.

ORB Run-time Scheduler: A real-time scheduler [44] maps application QoS requirements, such as include bounding end-to-end latency and meeting periodic scheduling deadlines, to ORB endsystem/network resources, such as ORB endsystem/network resources include CPU, memory, network connections, and storage devices. TAO's run-time scheduler supports both static [23] and dynamic [77] real-time scheduling strategies.

Real-time ORB Core: An ORB Core delivers client requests to the Object Adapter and returns responses (if any) to clients. TAO's real-time ORB Core [45] uses a multi-threaded, preemptive, priority-based connection and concurrency architecture [75] to provide an efficient and predictable CORBA IIOP protocol engine.

Real-time I/O subsystem: TAO's real-time I/O subsystem [17] extends support for CORBA into the OS. TAO's I/O subsystem assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be enforced. TAO also runs efficiently and relatively predictably on conventional I/O subsystems that lack advanced QoS features.

High-speed network interface: At the core of TAO's I/O subsystem is a "daisy-chained" network interface consisting of one or more ATM Port Interconnect Controller (APIC) chips [29]. APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps. In addition, TAO runs on conventional real-time interconnects, such as VME backplanes, multi-processor shared memory environments, as well as Internet protocols like TCP/IP.

TAO is developed atop lower-level middleware called ACE [78], which implements core concurrency and distribution patterns [49] for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications. ACE runs on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like Sun/Chorus ClassiX, LynxOS, and VxWorks.

B.2 Design Goals of TAO's Scheduling Service

To alleviate the limitations with existing scheduling strategies described in Section **??**, our research on CORBA real-time scheduling focuses on enabling applications to (1) *maximize total utilization*, (2) *preserve scheduling guarantees for critical operations* (when the set of critical operations can be identified), and (3) *adapt flexibly to different application and platform characteristics*. These three goals are illustrated in Figure 23 and summarized below:

Goal 1. Higher utilization: The upper pair of timelines in Figure 23 demonstrates our first research goal: *higher utilization*. This timeline shows a case where a critical operation execution did not, in fact, use its worst-case execution time. With dynamic scheduling, an additional non-critical operation could be dispatched, thereby achieving higher resource utilization.

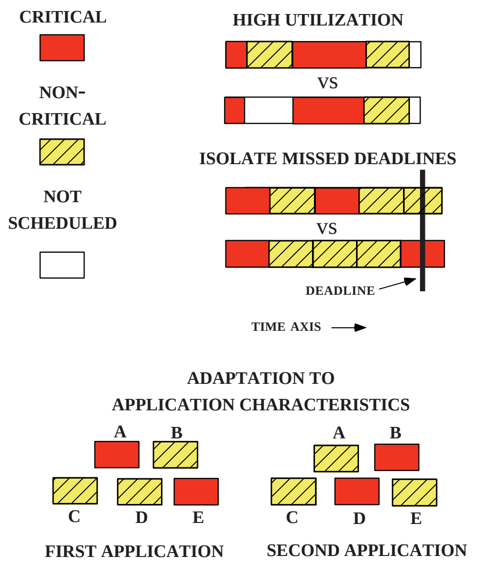

Figure 23: Design Goals of TAO's Dynamic Scheduling Service

Goal 2. Preserving scheduling guarantees: The lower pair of timelines in Figure 23 demonstrates our second research goal: *preserving scheduling guarantees for critical operations*. This timeline depicts a statically scheduled timeline, in which the worst-case execution time of the critical operation must be scheduled. In the lower timeline, priority is based on traditional scheduling parameters, such as rate and laxity. In the upper timeline, criticality is also included. Both timelines depict schedule overrun. When criticality is considered, only non-critical operations miss their deadlines.

Goal 3. Adaptive scheduling: The sets of operation blocks at the bottom of Figure 23 demonstrate our third research goal: *providing applications with the flexibility to adapt to varying application requirements and platform features*. In this example, the first and second applications use the same five operations. However, the first application considers operations A and E critical, whereas the second application considers operations B and D critical. By allowing applications to select which operations are critical, it should be possible to provide scheduling behavior that is appropriate to each application's individual requirements.

These goals motivate the design of TAO's strategized scheduling service framework, described in Section B.3. For the real-time systems [10, 23, 17, 45] that TAO has been applied to, it has been possible to identify a core set of operations whose execution before deadlines is *critical* to the integrity of the system. Therefore, the TAO's scheduling service is designed to ensure that critical CORBA operations will meet their deadlines, even when the total utilization exceeds the schedulable bound.

If it is possible to ensure deadlines will be met, then adding operations to the schedule to increase total CPU utilization will not increase the risk of missing deadlines. The risk will only increase for those operations whose execution prior to deadline is *not* critical to the integrity of the system. In this way, the risk to the whole system is minimized when it is loaded for higher utilization.

B.3 TAO's Strategized Scheduling Service Framework

TAO's scheduling service framework is designed to support a variety of scheduling strategies, including RMS, EDF, MLF, and MUF. This flexibility is achieved in TAO via the *Strategy* design pattern [49]. This pattern encapsulates a family of scheduling algorithms within a fixed interface. Within TAO's strategized scheduling service, the scheduling strategies themselves are interchangeable and can be varied independently.

The architecture and behavior of TAO's strategized scheduling service is illustrated in Figure 24. This architecture evolved from our earlier work on a CORBA scheduling service [23] that supported purely static rate monotonic scheduling. The steps involved in configuring and processing requests are described below. Steps 1-6 typically occur off-line during the schedule configuration process, whereas steps 7-10 occur on-line, underscoring the hybrid nature of TAO's scheduling architecture.

Step 1: A CORBA application specifies QoS information and passes it to TAO's scheduling service, which is implemented as a CORBA object, *i.e.*, it implements an IDL interface. The application specifies a set of values (`RT_Infos`) for the characteristics of each of its schedulable operations (`RT_Operations`). In addition, the application specifies invocation dependencies between these operations.

Step 2: At configuration time, which can occur either off-line or on-line, the application passes this QoS information into TAO's scheduling service via its *input interface*. TAO's scheduling service stores the QoS information in its repository of `RT_Info` descriptors. TAO's scheduling service's input interface is described further in Section B.5.1.

TAO's scheduling service constructs operation dependency graphs based on information registered with it by the application. The scheduling service then identifies threads of execution by examining the terminal nodes of these dependency graphs. Nodes that have outgoing edges but no incoming edges in the dependency graph are called *consumers*. Consumers are dispatched after the nodes on which they depend. Nodes that have incoming edges but no outgoing edges are called *suppliers*. Suppliers correspond to distinct threads of execution in the system. Nodes with incoming *and* outgoing edges can fulfill both roles.

Step 3: In this step, TAO's scheduling service assesses schedulability. A set of operations is considered *schedulable* if all operations in the critical set are guaranteed to meet their deadlines. Schedulability is assessed according to whether CPU utilization by operations in and above the minimum critical priority is less than or equal to the schedulable bound.

Step 4: Next, TAO's scheduling service assigns static priorities and subpriorities to operations. These values are assigned according to the specific strategy used to configure the scheduling service. For example, when the TAO scheduling service is configured with the MUF strategy, static priority is assigned according to operation criticality. Likewise, static subpriority is assigned according to operation importance and dependencies.

Step 5: Based on the specific strategy used to configure it, TAO's scheduling service divides the dispatching priority and dispatching subpriority components into statically and dynamically assigned portions. The static priority and static subpriority values are used to assign the static portions of the dispatching priority and dispatching subpriority of the operations. These dispatching priorities and subpriorities reside in TAO's `RT_Info` repository.

Step 6: Based on the assigned dispatching priorities, and in accordance with the specific strategy used to configure the off-line scheduling service, the number and types of dispatching queues needed to dispatch the generated schedule are assigned. For example, when the scheduling service is configured with the MLF strategy, there is a single queue, which uses laxity-based prioritization. As before, this configuration information resides in the `RT_Info` repository.

Step 7: At run-time start up, the configuration information in the `RT_Info` repository is used by the scheduling service's run-time scheduler component, which is collocated within an ORB endsystem. The ORB uses the run-time scheduler to retrieve (1) the thread priority at which each queue dispatches operations and (2) the type of dispatching prioritization used by each queue. The scheduling service's run-time component provides this information to the ORB via its *output interface*, as described in Section B.5.2.

Step 8: In this step, the ORB configures its *dispatching modules*, *i.e.*, the I/O subsystem, the ORB Core, and/or the Event Service. The information from the scheduling service's output interface is used to create the correct number and types of queues, and associate them with the correct thread priorities that service the queues. This configuration process is described further in Section B.5.3.

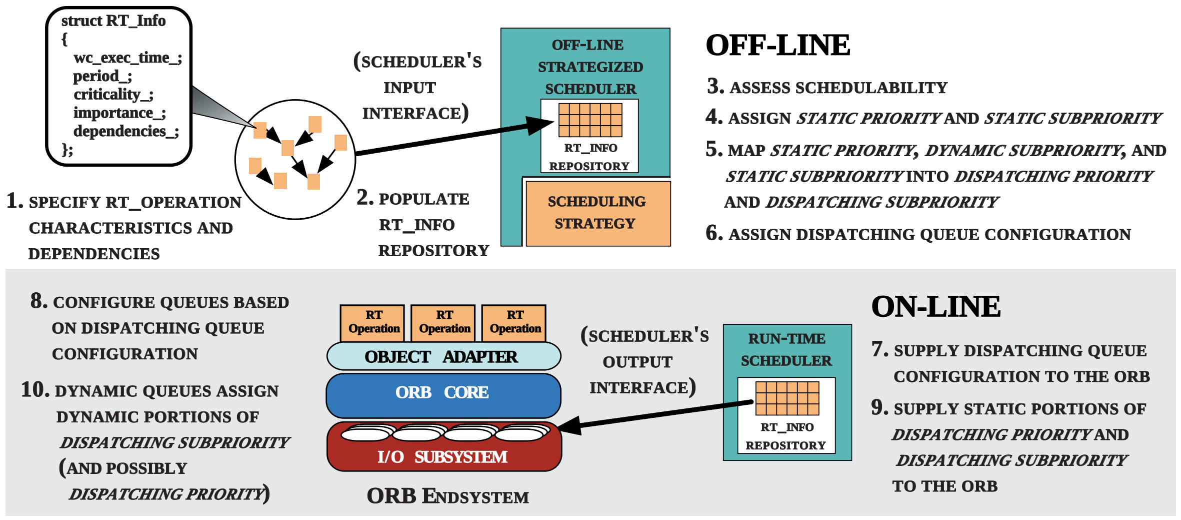
struct RT_Info
{
wc_exec_time_;
period_;
criticality_;
importance_;
dependencies_;
};
1. SPECIFY RT_OPERATION CHARACTERISTICS AND DEPENDENCIES
(SCHEDULER'S INPUT INTERFACE)
2. POPULATE RT_INFO REPOSITORY
OFF-LINE STRATEGIZED SCHEDULER
RT_INFO REPOSITORY
SCHEDULING STRATEGY
OFF-LINE
3. ASSESS SCHEDULABILITY
4. ASSIGN STATIC PRIORITY AND STATIC SUBPRIORITY
5. MAP STATIC PRIORITY, DYNAMIC SUBPRIORITY, AND STATIC SUBPRIORITY INTO DISPATCHING PRIORITY AND DISPATCHING SUBPRIORITY
6. ASSIGN DISPATCHING QUEUE CONFIGURATION
8. CONFIGURE QUEUES BASED ON DISPATCHING QUEUE CONFIGURATION
10. DYNAMIC QUEUES ASSIGN DYNAMIC PORTIONS OF DISPATCHING SUBPRIORITY (AND POSSIBLY DISPATCHING PRIORITY)
RT Operation
RT Operation
RT Operation
OBJECT ADAPTER
ORB CORE
I/O SUBSYSTEM
ORB ENDSYSTEM
(SCHEDULER'S OUTPUT INTERFACE)
RUN-TIME SCHEDULER
RT_INFO REPOSITORY
ON-LINE
7. SUPPLY DISPATCHING QUEUE CONFIGURATION TO THE ORB
9. SUPPLY STATIC PORTIONS OF DISPATCHING PRIORITY AND DISPATCHING SUBPRIORITY TO THE ORB

Figure 24: Processing Steps in TAO's Dynamic Scheduling Service Architecture

Step 9: When an operation request arrives from a client at run-time, the appropriate dispatching module must identify the dispatching queue to which the request belongs and initialize the request's dispatching subpriority. To accomplish this, the dispatching module queries TAO's scheduling service's output interface, as described in Section B.5.2. The run-time scheduler component of TAO's scheduling service first retrieves the static portions of the dispatching priority and dispatching subpriority from the `RT_Info` repository. It then supplies the dispatching priority and dispatching subpriority to the dispatching module.

Step 10: If the dispatching queue where the operation request is placed was configured as a *dynamic queue* in step 8, the dynamic portions of the request's dispatching subpriority (and possibly its dispatching priority) are assigned. This queue first does this when it enqueues the request. This queue then updates these dynamic portions as necessary when other operations are enqueued or dequeued.

The remainder of this section describes TAO's strategized scheduling service framework in detail. Section B.4 motivates why TAO allows applications to vary their scheduling strategy and Section B.5 shows how TAO's framework design achieves this flexibility.

B.4 Motivation for TAO's Strategized Scheduling Architecture

The flexibility of the architecture for TAO's strategized scheduling service is motivated by the following two goals:

1. *Shield application developers from unnecessary implementation details of alternative scheduling strategies* – This improves the system's reliability and maintainability, as described below.
2. *Decouple the strategy for priority assignment from the dispatching model so the two can be varied independently* – This increases the system's flexibility to adapt to varying application requirements and platform features.

TAO's scheduling strategy framework is designed to minimize unnecessary constraints on the values application developers specify to the input interface described in Section B.5.1. For instance, one (non-recommended) way to implement the RMS, EDF, and MLF strategies in TAO's scheduling service framework would be to implement them as variants of the MUF strategy. This can be done by manipulating the values of the operation characteristics [50]. However, this approach would tightly couple applications to the MUF scheduling strategy and the strategy being emulated.

There is a significant drawback to tightly coupling the behavior of a scheduling service to the characteristics of application operations. In particular, if the value of one operation characteristic used by the application changes, developers must remember to manually modify other operation characteristics specified to the scheduling service in order to preserve the same mapping. In general, we prefer to shield application developers from such unnecessary details.

To achieve this encapsulation, TAO's scheduling service allows applications to specify the entire set of possible operation characteristics using its input interface. In the scheduling strategies implemented in TAO, mappings between the input and output interfaces are entirely encapsulated within the strategies. Therefore, they need not require any unnecessary manipulation of input values. This decouples them from operation characteristics they need not consider.

Additional decoupling within the scheduling strategies themselves is also beneficial. Thus, each scheduling strategy in TAO specifies the following two distinct levels in its mapping from input interface to output interface:

1. Urgency assignment: The first level assigns *urgency* components, *i.e.*, static priority, dynamic subpriority, and static subpriority, based on (1) the operation characteristics specified to the input interface and (2) the selected scheduling strategy, *e.g.*, MUF, MLF, EDF, or RMS.

2. Dispatching (sub)priority assignment: The second level assigns dispatching priority and dispatching subpriority in the output interface based on the urgency components assigned in the first level.

By decoupling (1) the strategy for urgency assignment from (2) the assignment of urgency to dispatching priority and dispatching subpriority, TAO allows the scheduling strategy and the underlying dispatching model to vary independently. This decoupling allows a given scheduling strategy to be used on an OS that supports either preemptive or non-preemptive threading models, with only minor modification to the scheduling strategy. In addition, it facilitates comparison of scheduling strategies over a range of dispatching models, from fully preemptive-by-urgency, through preemptive-by-priority-band, to entirely non-preemptive. These models are discussed further in Section B.5.6.

B.5 Enhancing TAO's Scheduling Strategy Flexibility

The QoS requirements of applications and the hardware/software features of platforms and networks on which they are hosted often vary significantly. For instance, a scheduling strategy that is ideal for telecommunication call processing may be poorly suited for avionics mission computing [10]. Therefore, TAO's scheduling service framework is designed to allow applications to vary their scheduling strategies. TAO supports this flexibility by decoupling the *fixed* portion of its scheduling framework from the *variable* portion, as follows:

Fixed interfaces: The fixed portion of TAO's strategized scheduling service framework is defined by the following two interfaces:

- **Input Interface:** As discussed in Section B.5.1, the input interface consists of the three operations shown in Figure 25. Application can use these operations to manipulate QoS characteristics expressed with TAO's `RT_Info` descriptors [23] (steps 1 and 2 of Figure 24).

- **Output Interface:** As discussed in Section B.5.2, the output interface consists of the two operations shown in Figure 26. One operation returns the dispatching module configuration information (step 7 of Figure 24). The other returns the dispatching priority and dispatching subpriority components assigned to an operation (step 9 of Figure 24). Section B.5.3 describes how TAO's dispatching modules use information from TAO's scheduling service's output interface to configure and manage dispatching queues, as well as dispatch operations according to the generated schedule.

Variable mappings: The variable portion of TAO's scheduling service framework is implemented by the following two distinct mappings:

- **Input Mapping:** The input mapping assigns urgencies to operations according to the desired scheduling strategy. Section B.5.4 describes how each of the strategies implemented in TAO maps from the input interface to urgency values.

- **Output Mapping:** The output mapping assigns dispatching priority and dispatching subpriority according to the underlying dispatching model. Section B.5.5 describes how the output mapping translates the assigned urgency values into the appropriate dispatching priority and dispatching subpriority values for the output interface. Section B.5.6 describes alternatives to the output mapping used in TAO and discusses key design issues related to these alternatives.

The remainder of this section describes how TAO's scheduling service implements these fixed interfaces and variable mappings.

B.5.1 TAO's Scheduling Service Input Interface

As illustrated in steps 1 and 2 of Figure 24, applications use TAO's scheduling service input interface to convey QoS information that prioritizes operations. TAO's scheduling service input interface consists of the CORBA IDL interface operations shown in Figure 25 and outlined below.

create(): This operation takes a string with the operation name as an input parameter. It creates a new `RT_Info` descriptor for that operation name and returns a handle for that descriptor to the caller. If an `RT_Info` descriptor for that operation name already exists, `create` raises the DUPLICATE_NAME exception.

add_dependency(): This operation takes two `RT_Info` descriptor handles as input parameters. It places a dependency on the second handle's operation in the first handle's `RT_Info` descriptor. This dependency informs the scheduler that a flow of control passes from the second operation to the first. If either of the handles refers to an invalid `RT_Info` descriptor, `add_dependency` raises the UNKNOWN_TASK exception.

```
interface  Scheduler
{
     //...

     // Create a new RT_Info descriptor for entry_point
     handle_t create (  in string entry_point  )
          raises ( DUPLICATE_NAME );


     // Add dependency to handle's RT_Info descriptor
     void add_dependency ( in handle_t handle,
                           in handle_t dependency  )
          raises ( UNKNOWN_TASK );


      // Set values of operation characteristics
     // in handle's RT_Info descriptor
     void set ( in handle_t handle,
                in Criticality criticality,
                in Time worstcase_exec_time,
                in Period_period,
                in Importance importance  )
          raises ( UNKNOWN_TASK );

     //...
}
```

Figure 25: TAO Scheduling Service Input IDL Interface

set(): This operation takes an `RT_Info` descriptor handle and values for several operation characteristics as input parameters. The `set` operation assigns the values of operation characteristics in the handle's `RT_Info` descriptor to the passed input values. If the passed handle refers to an invalid `RT_Info` descriptor, `set` raises the UNKNOWN_TASK exception.

B.5.2 TAO's Scheduling Service Output Interface

The output interface for TAO's scheduling service consists of the CORBA IDL interface operations shown in Figure 26.

The first operation, `dispatch_configuration`, provides configuration information for queues in the dispatching modules used by the ORB endsystem (step 7 of Figure 24). It takes a dispatching priority value as an input parameter. It returns the OS thread priority and dispatching type corresponding to that dispatching priority level. The run-time scheduler component of TAO's scheduling service retrieves these values from the `RT_Info` repository, where they were stored by TAO's off-line scheduling component (step 6 of Figure 24).

The UNKNOWN_DISPATCH_PRIORITY exception will be raised if the `dispatch_configuration` operation is passed a dispatching priority that is not in the schedule. Likewise, if a schedule has not been generated, the `dispatch_configuration` operation raises the NOT_SCHEDULED exception.

The second operation, `priority`, provides dispatching priority and dispatching subpriority information for an operation request (step 9 of Figure 24). It takes an `RT_Info` descriptor handle as an input parameter and returns the assigned dispatching subpriority and dispatching priority as output parameters.

```
interface  Scheduler
{
     //...

     // Get configuration information for the queue that will dispatch all
     // RT_Operations that are assigned dispatching priority d_priority
     void dispatch_configuration ( in Dispatching_Priority d_priority,
                                   out OS_Priority os_priority,
                                   out Dispatching_Type d_type )
          raises ( UNKNOWN_DISPATCH_PRIORITY,
                   NOT_SCHEDULED );


      // Get static dispatching subpriority and dispatching
     //  priority assigned to the handle's RT_Operation
     void priority ( in handle_t handle,
                     out Dispatching_Subpriority d_subpriority,
                     out Dispatching_Priority d_priority)
          raises ( UNKNOWN_TASK,
                   NOT_SCHEDULED );

     //...
}
```

Figure 26: TAO Scheduling Service Output IDL Interface

The run-time component of TAO's scheduling service retrieves the dispatching priority and dispatching subpriority values stored in the `RT_Info` repository by its off-line component (step 5 of Figure 24). If the passed handle does not refer to a valid `RT_Info` descriptor, `priority` raises the UNKNOWN_TASK exception. If a schedule has not been generated, `priority` raises the NOT_SCHEDULED exception.

B.5.3 Integrating the TAO's Scheduling Service with Its Dispatching Modules

As noted in Section **??**, a key research challenge is to implement dispatching modules that can enforce end-to-end QoS requirements. This section (1) shows these dispatching modules fit within TAO's overall architecture, (2) describes the internal queueing mechanism of TAO's dispatching modules, and (3) discusses the issue of run-time control over dispatching priority within these dispatching modules.

Architectural placement: The output interface of TAO's scheduling service is designed to work with dispatching modules in any layer of the TAO architecture. For example, TAO's real-time extensions to the CORBA Event Service [10] uses the scheduler output interface, as does its I/O subsystem [17]. Figure 27(A) illustrates dispatching in TAO's real-time Event Service [10]. The client application pushes an event to TAO's Event Service. The Event Service's dispatching module enqueues events and dispatches them according to dispatching priority and then dispatching subpriority. Each dispatched

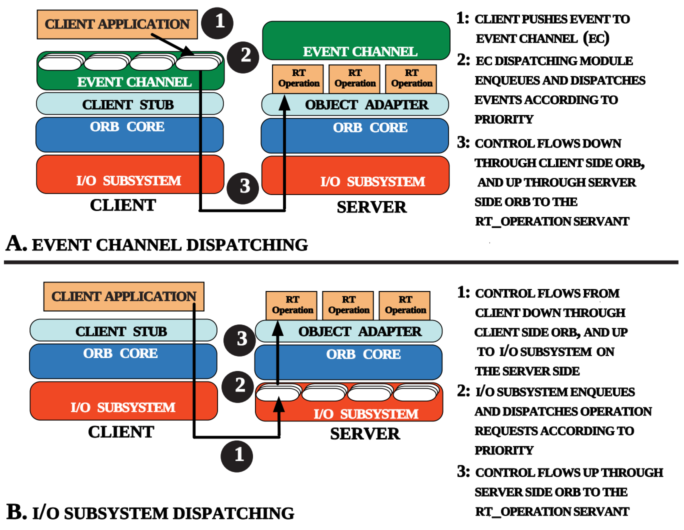
CLIENT APPLICATION
1
2
EVENT CHANNEL
EVENT CHANNEL
RT Operation
RT Operation
RT Operation
CLIENT STUB
OBJECT ADAPTER
ORB CORE
ORB CORE
I/O SUBSYSTEM
3
I/O SUBSYSTEM
CLIENT
SERVER
1: CLIENT PUSHES EVENT TO EVENT CHANNEL (EC)
2: EC DISPATCHING MODULE ENQUEUES AND DISPATCHES EVENTS ACCORDING TO PRIORITY
3: CONTROL FLOWS DOWN THROUGH CLIENT SIDE ORB, AND UP THROUGH SERVER SIDE ORB TO THE RT_OPERATION SERVANT
A. EVENT CHANNEL DISPATCHING
CLIENT APPLICATION
RT Operation
RT Operation
RT Operation
CLIENT STUB
3
OBJECT ADAPTER
ORB CORE
ORB CORE
2
I/O SUBSYSTEM
I/O SUBSYSTEM
CLIENT
SERVER
1
1: CONTROL FLOWS FROM CLIENT DOWN THROUGH CLIENT SIDE ORB, AND UP TO I/O SUBSYSTEM ON THE SERVER SIDE
2: I/O SUBSYSTEM ENQUEUES AND DISPATCHES OPERATION REQUESTS ACCORDING TO PRIORITY
3: CONTROL FLOWS UP THROUGH SERVER SIDE ORB TO THE RT_OPERATION SERVANT
B. I/O SUBSYSTEM DISPATCHING

Figure 27: Alternative Placement of Dispatching Modules

event results in a flow of control down through the ORB layers on the client and back up through the ORB layers on the server, where the operation is dispatched.

Figure 27(B) illustrates dispatching in TAO's I/O subsystem [17]. The client application makes direct operation calls to the ORB, which passes requests down through the ORB layers on the client and back up to the I/O subsystem layer on the server. The I/O subsystem's dispatching module enqueues operation requests and dispatches them according to their dispatching priority and dispatching subpriority, respectively. Each dispatched operation request results in a flow of control up through the higher ORB layers on the server, where the operation is dispatched.

Internal architecture: Figure 28 illustrates the general queueing mechanism used by the dispatching modules in TAO's ORB endsystem. In addition, this figure shows how the output information provided by TAO's scheduling service is used to configure and operate a dispatching module.

During system initialization, each dispatching module obtains the thread priority and dispatching type for each of its queues from the scheduling service's output interface, as described in Section B.5.2. Next, each queue is assigned a unique dispatching priority number, a unique thread priority, and an enumerated dispatching type. Finally, each dispatching module has an ordered queue of pending dispatches per dispatching priority.

To preserve QoS guarantees, operations are inserted into the appropriate dispatching queue according to their assigned dispatching priority. Operations within a dispatching queue are ordered by their assigned dispatching subpriority. To minimize priority inversions, operations are dispatched from the queue with the highest thread priority, preempting any operation executing in a lower priority thread [10]. To minimize preemption overhead, there is no preemption within a given priority queue.

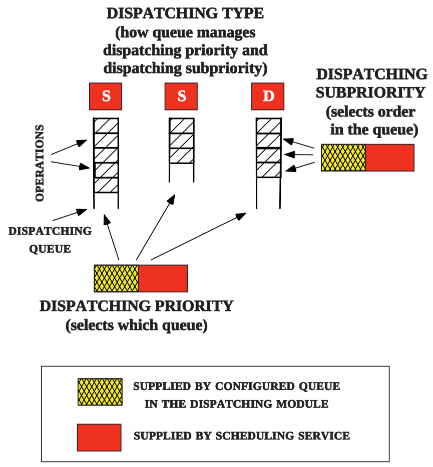
DISPATCHING TYPE
(how queue manages dispatching priority and dispatching subpriority)
S
S
D
DISPATCHING SUBPRIORITY
(selects order in the queue)
OPERATIONS
DISPATCHING QUEUE
DISPATCHING PRIORITY
(selects which queue)
SUPPLIED BY CONFIGURED QUEUE IN THE DISPATCHING MODULE
SUPPLIED BY SCHEDULING SERVICE

Figure 28: Example Queueing Mechanism in a TAO Dispatching Module

The following three values are defined for the dispatching type:

STATIC_DISPATCHING: This type specifies a queue that only considers the static portion of an operation's dispatching subpriority.

DEADLINE_DISPATCHING: This type specifies a queue that considers the dynamic and static portions of an operation's dispatching subpriority, and updates the dynamic portion according to the time remaining until the operation's deadline.

LAXITY_DISPATCHING: This type specifies a queue that considers the dynamic and static portions of an operation's dispatching subpriority, and updates the dynamic portion according to the operation's laxity.

The deadline- and laxity-based queues update operation dispatching subpriorities whenever an operation is enqueued or dequeued.

Run-time dispatching priority: Run-time control over dispatching priority can be used to achieve the preemptive-by-urgency dispatching model discussed in Section B.5.6. However, this model incurs greater complexity in the dispatching module implementation, which increases run-time overhead. Therefore, once an operation is enqueued in TAO's dispatching modules, none of the queues specified by the above dispatching types exerts control over an operation's dispatching priority at run-time.

As noted in Section B.5.5, all the strategies implemented in TAO map static priority directly into dispatching priority.

Compared with strategies that modify an operation's dispatching priority dynamically, this mapping simplifies the dispatching module implementation since queues need not maintain references to one another or perform locking to move messages between queues. In addition, TAO's strategy implementations also minimize run-time overhead since none of the queues specified by its dispatching types update any dynamic portion of an operation's dispatching priority. These characteristics meet the requirements of real-time avionics systems to which TAO has been applied [47, 10, 23, 45].

It is possible, however, for an application to define strategies that *do* modify an operation's dispatching priority dynamically. A potential implementation of this is to add a new constant to the enumerated dispatching types. In addition, an appropriate kind of queue must be implemented and used to configure the dispatching module according to the new dispatching type. Supporting this extension is simplified by the flexible design of TAO's scheduling service framework.

B.5.4 Input Mappings Implemented in TAO's Scheduling Service

In each of TAO's scheduling strategies, an input mapping assigns urgency to an operation according to a specific scheduling strategy. Input mappings for MUF, MLF, EDF, and RMS have been implemented in TAO's strategized scheduling service. Below, we outline each mapping.

In each mapping, static subpriority is assigned first using importance and second using a topological ordering based on dependencies. The canonical definitions of MLF, EDF, and RMS do not include a minimal static ordering. Adding it to TAO's strategy implementations for these strategies has no adverse effect, however. This is because MLF, EDF, and RMS require that *all* operations are guaranteed to meet their deadlines for the schedule to be feasible, under *any* ordering of operations with otherwise identical priorities. Moreover, static ordering has the benefit of ensuring determinism for each possible assignment of urgency values.

MUF mapping: The mapping from operation characteristics onto urgency for MUF is shown in Figure 29. Static priority is assigned according to criticality in this mapping. There are only two static priorities since we use only two criticality levels in TAO's MUF implementation. The critical set in this version of MUF is the set of operations that were assigned the *high* criticality value.

When MUF is implemented with only two criticality levels, the minimum critical priority is the static priority corresponding to the high criticality value. In the more general version of MUF [50], in which multiple criticality levels are possible, the critical set may span multiple criticality levels.

Dynamic subpriority is assigned in the MUF input mapping

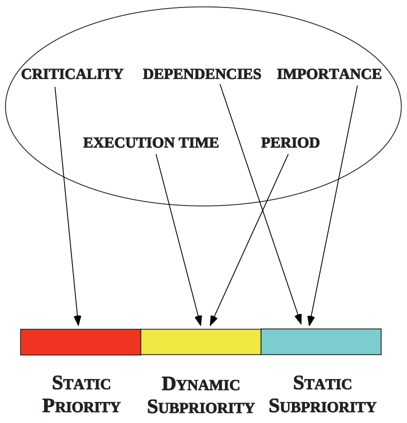

Figure 29: MUF Input Mapping

according to *laxity*. Laxity is a function of the operation's period, execution time, arrival time, and the time of evaluation.

MLF mapping: The MLF mapping shown in Figure 30 assigns a constant (zero) value to the static priority of each operation. This results in a single static priority. The minimum

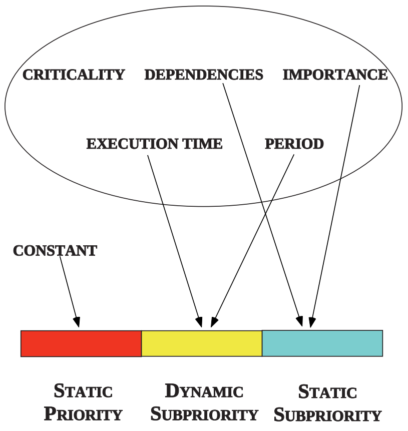

Figure 30: MLF Input Mapping

critical priority is this lone static priority. The MLF strategy assigns the dynamic subpriority of each operation according to its laxity.

EDF mapping: The EDF mapping shown in Figure 31 also assigns a constant (zero) value to the static priority of each operation. Moreover, the EDF strategy assigns the dynamic subpriority of each operation according to its *time-to-deadline*, which is a function of its period, its arrival time, and the time of evaluation.

RMS mapping: The RMS mapping shown in Figure 32 assigns the static priority of each operation according to its *period*, with higher static priority for each shorter period. The period for aperiodic execution must be assumed to be the worst

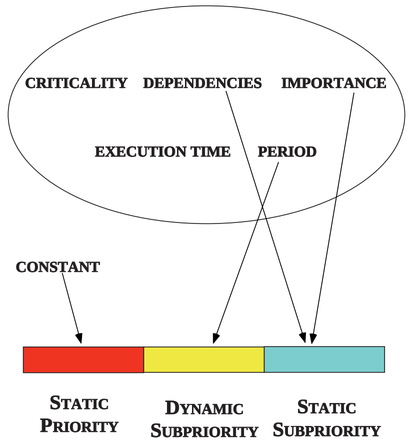

Figure 31: EDF Input Mapping

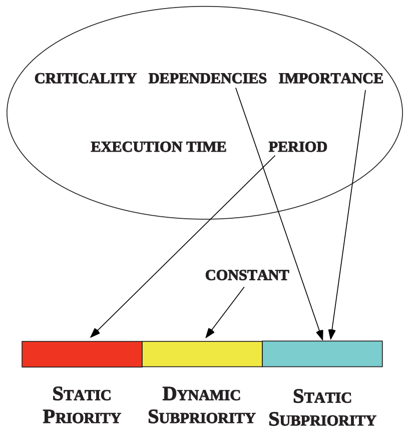

Figure 32: RMS Input Mapping

case. In RMS, all operations are critical, so the minimum critical priority is the minimum static priority in the system. The RMS strategy assigns a constant (zero) value to the dynamic subpriority of each operation.

This section explored the well known RMS, EDF, MLF, and MUF priority mappings. These mappings reflect opposing design forces of commonality and difference. TAO's strategized scheduling service leverages the commonality among these mappings to make its implementation more uniform. The differences between these mappings provide hot spots for adaptation to the requirements of specific applications.

B.5.5 Output Mapping Implemented in TAO's Scheduling Service

The need to correctly specify enforcable end-to-end QoS requirements for different operations motivates both the input and output mappings in TAO's strategized scheduling service. The input mappings described in Section B.5.4 specify priorities and subpriorities for operations. However, there is no mechanism to enforce these priorities, independent of the specific OS platform dispatching models. In each of TAO's scheduling strategies, an output mapping transforms these priority and subpriority values into dispatching priority and subpriority requirements that can be enforced by the specific dispatching models in real systems.

As described in Section B.5.3, operations are distributed to priority dispatching queues in the ORB according to their assigned dispatching priority. Operations are ordered within priority dispatching queues according to their designated dispatching subpriority. The scheduling strategy's output mapping assigns dispatching priority and dispatching subpriority to operations as a function of the urgency values specified by the scheduling strategy's input mapping.

Figure 33 illustrates the output mapping used by the scheduling strategies implemented in TAO. Each mapping is

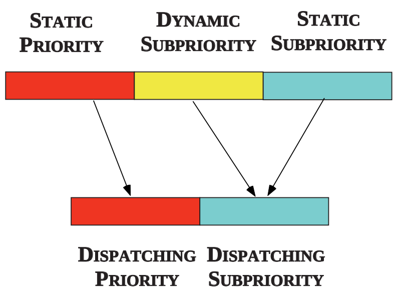

Figure 33: Output Mapping Implemented in TAO

described below.

Dispatching Priority: In this mapping, static priority maps directly to dispatching priority. This mapping corresponds to the priority band dispatching model described in Section B.5.2. Each unique static priority assigned by the input mapping results in a distinct thread priority in TAO's ORB request dispatching module.

Dispatching Subpriority: Dynamic subpriority and static subpriority map to dispatching subpriority. TAO's strategized scheduling service performs this mapping efficiently at run-time by transforming both dynamic and static subpriorities into a flat binary representation. A binary integer format of length k bits is used to store the dispatching subpriority value.

Because the range of dynamic subpriority values and the number of static subpriorities are known prior to run-time, a fixed number of bits can be reserved for each. Dynamic subpriority is stored in the m highest order bits, where $m = \lceil \lg(ds) \rceil$, and ds is the number of possible dynamic subpriorities. Static subpriority is stored in the next n lower order bits, where $n = \lceil \lg(ss) \rceil$, and ss is the number of static subpriorities.

TAO's preemption subpriority mapping scheme preserves the ordering of operation dispatches according to their assigned *urgency* values. Static subpriorities correspond to thread priorities. Thus, an operation with higher static priority will always preempt one with lower static subpriority. Operations with the same static priority are ordered first by dynamic subpriority and second by static subpriority.

B.5.6 Alternative Output Mappings

It is useful to consider the consequences of the specific output mapping described in Section B.5.5 and to evaluate the uses and implications of alternative output mappings. The scheduling strategies implemented in TAO strike a balance between preemption granularity and run-time overhead. This design is appropriate for the hard real-time avionics applications we have developed.

However, TAO's strategized scheduling architecture is designed to adapt to the needs of a range of applications, not just hard real-time avionics systems. Different types of applications and platforms may require different resolutions of key design forces.

For example, an application may run on a platform that *does not* support preemptive multi-threading. Likewise, other platforms do not support thread preemption and multiple thread priority levels. In such cases, TAO's scheduling service framework assigns all operations the same constant dispatching priority and maps the entire urgency tuple directly into the dispatching subpriority [50]. This mapping correctly assigns dispatching priorities and dispatching subpriorities for a non-preemptive dispatching model. On a platform without preemptive multi-threading, the application could thus dispatch all operations in a single thread of execution, from a single priority queue.

Another application might run on a platform that *does* support preemptive multi-threading and a large number of distinct thread priorities. Where thread preemption and a very large number of thread priorities are supported, one alternative is a dispatching model that is preemptive by *urgency*. This design may incur higher run-time overhead, but can allow finer preemption granularity. The application in this second example might accept the additional time and space overhead needed to preemptively dispatch operations by urgency, in exchange for reducing the amount of priority inversion incurred by the dispatching module.

Depending on (1) whether the OS supports thread preemption, (2) the number of distinct thread priorities supported, and (3) the preemption granularity desired by the application, several dispatching models can be supported by the output interface of TAO's scheduling service. Below, we examine three canonical variations supported by TAO, which are illustrated in Figure 34.

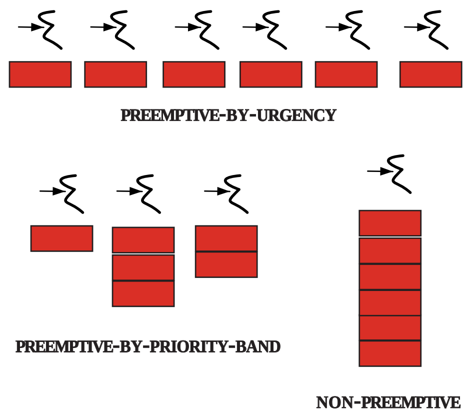

Figure 34: Dispatching Models supported by TAO

Preemptive-by-urgency: One consequence of the input and output mappings implemented in TAO is that the purely dynamic EDF and MLF strategies are non-preemptive. Thus, a newly arrived operation will not be dispatched until the operation currently executing has run to completion, even if the new operation has greater urgency. By assigning dispatching priority according to urgency, all scheduling strategies can be made fully preemptive.

This dispatching model maintains the invariant that the highest urgency operation that is able to execute is executing at any given instant, modulo the OS dispatch latency overhead [14]. This model can be implemented only on platforms that (1) support fully preemptive multitasking and (2) provide at least as many distinct real-time thread priorities as the number of distinct operation urgencies possible in the application.

The preemptive-by-urgency dispatching model can achieve very fine-grained control over priority inversions incurred by the dispatching modules. This design potentially reduces the time bound of an inversion to that for a thread context switch plus any switching overhead introduced by the dispatching mechanism itself. Preemptive-by-urgency achieves its precision at the cost of increased time and space overhead, however. Although this overhead can be reduced for applications whose operations are known in advance, using techniques like perfect hashing [56], overhead from additional context switches will still be incurred.

Preemptive-by-priority-band: This model divides the range of all possible urgencies into fixed priority bands. It is similar to the non-preemptive dispatching model used by message queues in the UNIX System V STREAMS I/O subsystem [79, 17]. This dispatching model maintains a slightly weaker invariant than the preemptive-by-urgency model. At any given instant, an operation from the highest fixed-priority band that has operations able to execute is executing.

This dispatching model requires thread preemption and at least a small number of distinct thread priority levels. These features are now present in many operating systems. The preemptive-by-priority-band model is a reasonable choice when it is desirable or necessary to restrain the number of distinct preemption levels.

For example, a dynamic scheduling strategy can produce a large number of distinct urgency values. These values must be constrained on operating systems, such as like Windows NT [80], that support only a small range of distinct thread priorities. Operations in the queue are ordered by a subpriority function based on urgency. The strategies implemented TAO's strategized scheduling service use a form of this model, as described in Section B.5.5.

Non-preemptive: This model uses a single priority queue and is non-preemptive. It maintains a still weaker invariant: the operation executing at any instant had the greatest urgency at the time of last dispatch. As before, operations are ordered according to their urgency within the single dispatching queue. Unlike the previous models, however, this model can be used on platforms that lack thread preemption or multi-threading.