TAO: a High-performance ORB Endsystem Architecture for Real-time CORBA

Douglas C. Schmidt, Aniruddha Gokhale, Timothy H. Harrison, David Levine, and Chris Cleeland

{schmidt,gokhale,harrison,levine,cleeland}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

This document has been submitted as an RFI response to the OMG Special Interest Group on Real-time CORBA.

Abstract

Many application domains (such as avionics, telecommunications, and multimedia) require real-time guarantees from the underlying networks, operating systems, and middleware components to achieve their quality of service (QoS) requirements. In addition to providing end-to-end QoS guarantees, applications in these domains must be flexible and reusable. Requirements for flexibility and reusability motivate the use of object-oriented middleware like the Common Object Request Broker Architecture (CORBA). However, the performance levels, and QoS enforcement features of current CORBA implementations are not yet suited for applications with hard real-time requirements (e.g., avionics mission computers) and statistical real-time requirements (e.g., teleconferencing).

Although some operating systems, networks, and protocols now support real-time scheduling, though they do not provide integrated solutions. This paper describes the architectural features and optimizations we are developing for real-time ORB endsystems that can deliver end-to-end QoS guarantees to applications. The main thrust of this paper is that advances in real-time distributed object computing can be achieved only by systematically pinpointing performance bottlenecks; optimizing the performance of networks, ORB endsystems, common services, and applications; and simultaneously integrating techniques and tools that simplify application development.

1 Introduction

An increasingly important class of distributed applications requires stringent quality of service (QoS) guarantees. These applications include telecommunication systems (*e.g.,* call processing and switching), avionics control systems (*e.g.,* operational flight programs for fighter aircraft), multimedia (*e.g.,* video-on-demand and teleconferencing), and simulations (*e.g.,* battle readiness planning). In addition to requiring QoS guarantees, these applications must be flexible and reusable.

The Common Object Request Broker Architecture (CORBA) is a distributed object computing middleware standard defined by the Object Management Group (OMG) [1]. CORBA is intended to support the production of flexible and reusable distributed services and applications. Many implementations of CORBA are now available.

1.1 Limitations of CORBA for Real-time Applications

Our experience using CORBA on telecommunication [2], avionics [3], and medical projects [4] indicates that it is well-suited for request/response applications. However, CORBA is not yet suited for performance-sensitive, real-time applications for the following reasons:

• **Lack of QoS specification and enforcement:** CORBA 2.0 does not define APIs for specifying end-to-end QoS requirements. Likewise, existing CORBA implementations do not provide support for end-to-end QoS enforcement, *i.e.,* from application-to-application across a network. For instance, there is no standard way for clients to indicate to an ORB the relative priorities of their requests. Likewise, there are no means for clients to inform an ORB how frequently to execute operations that must run periodically.

• **Lack of real-time features:** CORBA 2.0 does not define key features that are necessary to support real-time programming. For instance, although the CORBA inter-operability protocol (GIOP) supports asynchronous messaging, there is no standard programming language mapping for transmitting ORB requests asynchronously. Likewise, the CORBA 2.0 specification does not require an ORB to notify clients when transport layer flow control occurs, nor does it support timed operations [**?**]. As a result, it is hard to write portable and efficient real-time applications that are guaranteed not to block indefinitely when ORB endsystem and network resources are temporarily unavailable.

• **Lack of performance optimizations:** Existing ORBs incur significant throughput [5, 6, 4] and latency [7, 8] overhead. These overheads stem from excessive data copying, non-optimized presentation layer conversions, internal message buffering strategies that produce non-uniform behavior for different message sizes, inefficient demultiplexing algorithms, long chains of intra-ORB virtual method calls, and lack of integration with underlying real-time OS and network QoS mechanisms.

1.2 Overcoming the Limitations with CORBA

This paper describes how we are overcoming these limitations via a high-performance, real-time ORB called TAO (The ACE ORB) [3, 8, 9]. TAO is a multi-threaded ORB endsystem that runs on a range of OS platforms that support real-time features including Solaris, Windows NT, and VxWorks. Our objective in developing TAO is to identify the key architectural patterns and performance optimizations necessary to build high-performance, real-time ORBs. We are transferring this technology to ORB vendors who will productize and commercialize our research results.

In addition to providing a real-time ORB, TAO is an integrated *ORB endsystem architecture* that consists of a high-performance I/O subsystem [10, 11] and an ATM Port Interconnect Controller (APIC) [12]. This ORB endsystem architecture is designed to support end-to-end gigabit data rates and ∼10 msec latency to CORBA applications. In addition, TAO is designed with a layered architecture, which can also run on embedded platforms that are linked via real-time interconnects (such as 1553 buses). Figure 12 illustrates the primary components in TAO.

QoS guarantees supported by TAO can be both *deterministic* (*e.g.,* for hard real-time avionics applications where meeting periodic deadlines is crucial) and *statistical* (*e.g.,* for latency constrained applications like teleconference and video-on-demand where minor fluctuations in scheduling and reliability guarantees are tolerated). Because hard-real time systems are more amenable to deterministic analysis, the work presented in this paper focuses on them. Consequently, while the architectural features and optimizations discussed in this paper are applicable to statistical real-time systems, (including constrained latency multimedia systems), most of the scheduling discussion centers on hard real-time systems.

This paper is organized as follows: Section 2 outlines the features and optimizations required by ORB endsystems to provide end-to-end QoS guarantees to applications and higher layer services (such as Events, Logging, and Streaming [13]); Section 3 describes the feature enhancements and optimizations we are developing for TAO; Section 4 discusses the TAO real-time Scheduling Service in detail; Section 5 discusses how we have applied TAO to develop a Real-time CORBA Event Service that provides a mechanism for dispatching asynchronous events according to the periodic processing requirements of event consumers; and Section 6 presents concluding remarks and describes the current status of TAO; and Appendix A presents a biography of the TAO project PI, Dr. Douglas C. Schmidt, as well as other members of the TAO group.

2 Endsystem Requirements for Real-time Object Request Brokers

This section outlines the endsystem requirements for real-time Object Request Brokers (ORBs). For completeness, an overview of the primary components in the CORBA reference model is presented first.

2.1 Components in the CORBA Model

CORBA ORBs allow clients to invoke operations on target object implementations without concern for where the object resides, what language the object is written in, what OS/hardware platform it runs on, or what communication protocols and networks are used to interconnect distributed objects [14]. To support this level of transparency, the CORBA reference model defines the following components (shown in Figure 1):

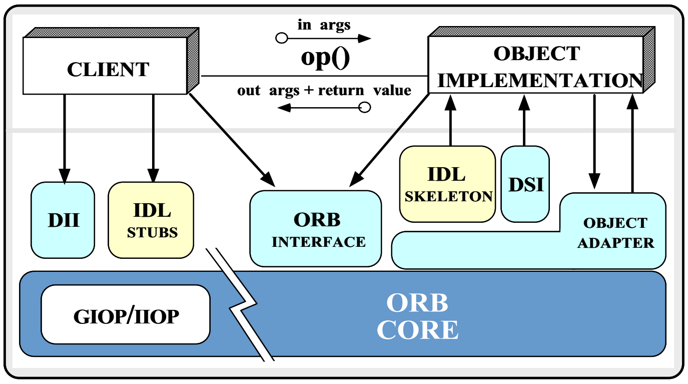

Figure 1: Components in the CORBA Reference Model

• **Object Implementation:** This defines operations that implement an interface specified using the CORBA Interface Definition Language (IDL). Object implementations can be written in a variety of languages including C, C++, Java, Smalltalk, and Ada.

• **Client:** This program entity invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the client. Ideally, it should be as simple as calling an operation on an object, *i.e.,* obj−>op(args). The remaining components in Figure 1 help to support this level of transparency.

• **Object Request Broker (ORB) Core:** When a client invokes an operation, the ORB Core is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning a response (if any) to the client. ORBs that comply with the CORBA 2.0 specification use implementations of the General Inter-ORB Protocol (GIOP) to transport client requests, replies, and error messages. GIOP defines an on-the-wire format, known as the Common Data Representation (CDR), for each object data type. In addition, GIOP defines the sequence of protocol messages that implement inter-ORB communication. The Internet Inter-ORB Protocol (IIOP) is a mapping of GIOP for TCP/IP protocols. IIOP is emerging as the standard communication protocol for distributed object computing over the Internet.

• **ORB Interface:** An ORB is a logical entity that may be implemented in various ways (such as one or more processes/threads or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper methods such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

• **OMG IDL stubs and skeletons:** OMG IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. The transformation between OMG IDL definitions and the target programming language is automated by a OMG IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations of stub and skeleton code.

• **Dynamic Invocation Interface (DII):** This interface allows clients to directly access the underlying request mechanisms provided by an ORB. Clients can use the DII to dynamically issue requests to target objects without requiring IDL interface-specific stubs to be linked in to applications. Unlike IDL stubs (which only allow twoway and oneway requests), the DII also allows clients to make non-blocking *deferred synchronous* calls, which separate send operations from receive operations.

• **Dynamic Skeleton Interface (DSI):** This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not know at compile-time the type of object it is implementing. The client making the request has no idea whether the implementation uses the type-specific IDL skeletons or the dynamic skeletons.

• **Object Adapter:** This assists the ORB by demultiplexing requests to the target object and dispatching operation upcalls on the object. In addition, the Object Adapter associates object implementations with the ORB. Object Adapters can be specialized to provide support for certain object implementation styles (such as OODB Object Adapters for persistence, library Object Adapters for non-remote objects, and real-time Object Adapters for applications that require QoS guarantees).

2.2 Issues for High Performance, Real-Time CORBA

Real-time ORB endsystems require much more than CORBA middleware – they must also integrate with network adapters, operating system I/O subsystems, communication protocols, and common object services. Section 3 describes how these components are integrated into TAO. The remainder of this section outlines the requirements of real-time (RT) ORB endsystems and summarizes the material presented in subsequent sections.

• **Policies and mechanisms for specifying end-to-end application QoS requirements:** Real-time ORB endsystems must allow applications to specify the QoS requirements of their IDL operations using a small number of application-centric parameters (such as computation time, execution period, bandwidth and delay requirements). For instance, video-conferencing groupware may require high throughput with statistical real-time latency deadlines. In contrast, an operational flight control platform for avionics may require periodic processing with strict real-time deadlines.

QoS specification is not addressed by the current CORBA specification. Section 3.3 outlines how TAO allows applications to specify their QoS requirements using a combination of OMG IDL and an ORB QoS interface.

• **QoS enforcement from real-time operating systems and networks:** Regardless of the ability to specify QoS requirements, RT ORBs 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, storage throughput, network adapter throughput, and network connection bandwidth and latency. For instance, OS scheduling mechanisms must allow high priority client requests to run to completion and prevent them from being blocked indefinitely by lower priority requests [11, 15].

Section 3.1 describes the OS I/O subsystem and network adapter integrated with TAO. This infrastructure is designed to support end-to-end gigabit data rates and $\sim$10 msec latency to CORBA applications. In addition, TAO is designed to be layered atop other OS/network platforms (such as 1553 buses, VME backplanes, and multi-processor shared memory environments).

• **Optimized real-time communication protocols:** The throughput, latency, and reliability requirements of multimedia applications like teleconferencing are more stringent and varied 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 range of communication protocols that can be customized and optimized for specific application requirements and network/host environments.

Section 3.2 outlines TAO's real-time enhancements and optimizations to the CORBA General Inter-ORB Protocol (GIOP) [1], which specifies the request format and transmission protocol that enables interoperability among heterogeneous ORBs.

• **Optimized real-time request demultiplexing and dispatching:** ORB endsystems must demultiplex and dispatch incoming client requests to the appropriate operation of the target object. In conventional ORBs, demultiplexing occurs at multiple layers (*e.g.,* the network interface, the protocol stack, the user/kernel boundary, and the ORB's Object Adapter).

Layered demultiplexing is inappropriate for high-performance and real-time applications [16] because it reduces performance by increasing the number of times that internal tables must be searched while incoming client requests traverse various protocol processing layers. Likewise, layered demultiplexing can cause priority inversions because important target object-level QoS information is inaccessible to the lowest level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem.

Section 3.5 outlines TAO's de-layered active demultiplexing mechanism [17] and real-time dispatching mechanism [3], which can process client requests efficiently and predictably regardless of the number of active connections, application-level target object implementations, and operations defined in IDL interfaces.

• **Optimized memory management:** On modern RISC hardware, data copying consumes a significant amount of CPU, memory, and I/O bus resources. Therefore, multiple layers in an ORB endsystem (*e.g.,* the network adapters, I/O subsystem protocol stacks, Object Adapter, and presentation layer) must collaborate to minimize data copying [18].

Section 3.1 outlines TAO's zero-copy memory management mechanism, which behaves predictably and efficiently irrespective of user buffer sizes and endsystem workload.

• **Optimized presentation layer:** Presentation layer conversions transform application-level data into a portable format that masks byte order, alignment, and word length differences. There are many optimizations that reduce the cost of presentation layer conversions. For instance, [19] describes the tradeoffs between using compiled versus interpreted code for presentation layer conversions. Compiled marshaling code is efficient, but requires excessive amounts of memory, which is problematic in many embedded real-time environments. In contrast, interpreted marshaling code is slower, but more compact.

Section 3.6 outlines how TAO supports worst-case guarantees for both interpreted and compiled marshaling operations via a highly optimized protocol engine based on innovative compiler techniques and optimization principles. 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; and passing information between layers [20].

It is important to recognize that requirements for high performance may conflict with requirements for real-time determinism. For instance, real-time scheduling policies often rely on the predictability of endsystem operations like thread scheduling, demultiplexing, and message buffering. However, certain optimizations (such as using self-organizing search structures to demultiplex client requests) can increase the average-case performance of operations, while decreasing the predictability of any given operation. Therefore, TAO is designed with an open architecture that allows applications to select the appropriate tradeoffs between average-case and worst-case performance. Moreover, where possible, we use algorithms and data structures that can optimize for both performance *and* predictability. For instance, de-layered active demultiplexing [17] can increase ORB performance and predictability by eliminating excessive searching and avoiding priority inversions.

3 TAO: a High-performance ORB Endsystem for Real-Time CORBA

This section describes TAO (The ACE ORB), which is a high-performance ORB endsystem for real-time CORBA that we are developing at Washington University. This section describes the key layers and components in TAO (shown in Figure 2). Section 4 focuses on TAO's support for real-time scheduling in detail.

TAO is an ORB Endsystem Architecture that contains the following network interface, operating system, communication protocol, and CORBA middleware mechanisms:

- *Gigabit I/O subsystem* – which optimizes conventional OS I/O subsystems to execute at Gigabit rates over high-speed ATM networks;
- *Real-time GIOP protocol* – which provides efficient and predictable transmission of requests using enhancements to standard CORBA interoperability protocols;
- *QoS specification* – which allows applications and higher-level CORBA services to define end-to-end QoS parameters on a per-request and/or per-session basis;
- *Real-time Scheduling Service* – which determines the priority of client requests with hard real-time deadlines;
- *Real-time Object Adapter* – which demultiplexes and dispatches client requests according to real-time requirements; and
- *Presentation layer components* – which optimize key sources of marshaling, demarshaling, and data copying overhead in conventional ORBs.

3.1 Gigabit I/O Subsystem

To guarantee the QoS needs of applications, TAO requires guarantees from the underlying I/O subsystem. To accomplish this task, we are developing a high-performance network I/O subsystem. The components of this subsystem include (1) the network adapter, (2) a real-time protocol processing subsystem, and (3) the OS-level scheduler. Our research on TAO addresses all three components, as described below.

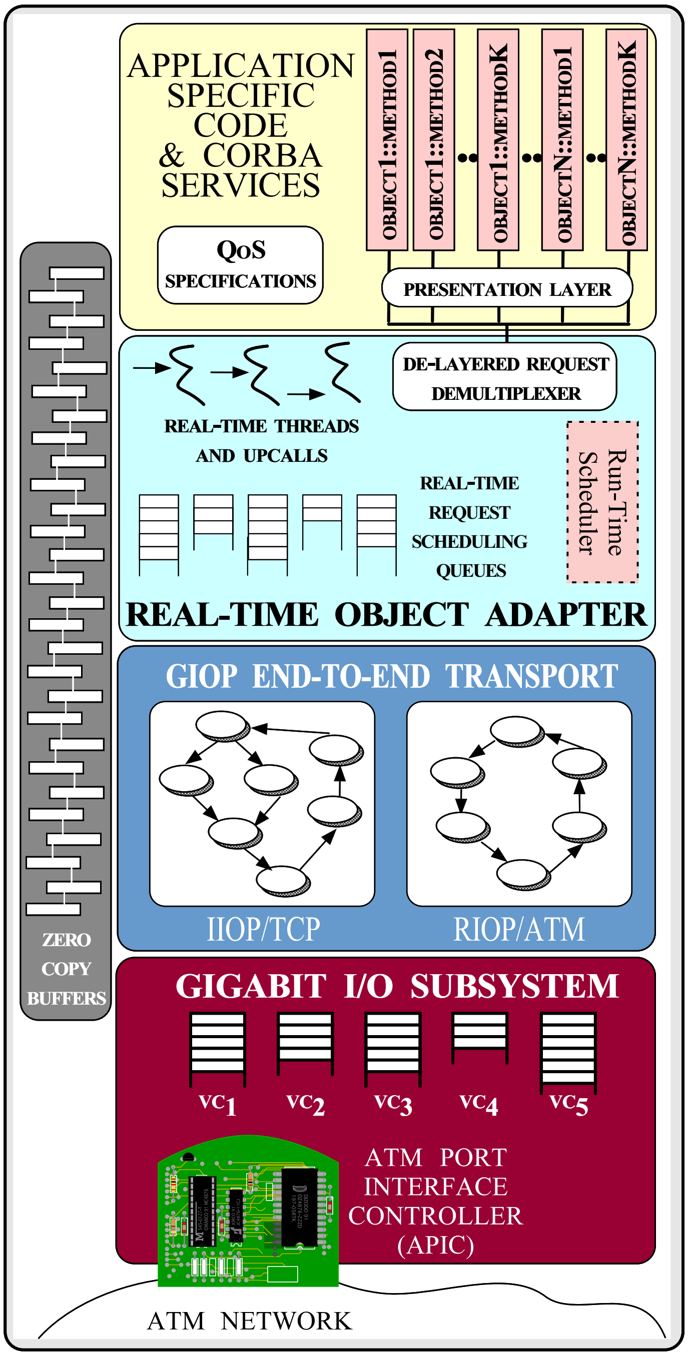

Figure 2: Architectural Layers in the TAO Real-time ORB Endsystem

• **High-speed network adapter:** At the heart of our I/O subsystem is a daisy-chained interconnect comprising a number of ATM Port Interconnect Controller (APIC) chips [21]. APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps.

• **Zero-copy buffer management:** Our Gigabit I/O subsystem builds on the APIC to enhance conventional operating systems with a zero-copy buffer management system. At the device level, the APIC interfaces directly with the main system bus and other I/O devices to transfer client requests between endsystem buffer pools and ATM virtual circuits without incurring additional data copying. The buffer pools for I/O devices support "direct demultiplexing" [10, 11, 12] of periodic and aperiodic client requests into memory shared among user- and kernel-resident threads.

• **Real-time I/O subsystem:** Although some operating systems (such as Solaris and Windows NT) now support real-time scheduling, they do not provide integrated solutions. For instance, Solaris 2.x provides a real-time scheduling class, which provides worst-case guarantees on the time required to dispatch threads who are in this thread class [15]. Threads in the Solaris real-time scheduling class run at fixed priorities that are always greater than time-shared or system threads.

The Solaris real-time scheduling class is relatively low-level, however, and does not support application-level QoS specifications. In addition, it is not integrated seemlessly with the Solaris STREAMS network I/O mechanism. In particular, the Solaris STREAMS implementation does not support QoS guarantees since STREAMS processing is performed at system threads priority, which is lower than all real-time threads. Therefore, the Solaris I/O subsystem is prone to priority inversions, where a high-priority real-time threads block indefinitely waiting for low-priority real-time threads to finish. Unbounded priority inversion is highly undesirable in most real-time environments.

We are enhancing the Solaris scheduling mechanism to support QoS guarantees by adding a new scheduling class, RT I/O, that supports periodic real-time applications. Once a thread of the RT I/O class is admitted by the OS, the scheduler 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.

The goal of the RT I/O scheduling class is to allow applications to specify their requirements in an intuitive manner. For instance, one implementation of the RT I/O scheduling class is based on Rate Monotonic scheduling, where applications can specify their processing requirements in terms of computation time C and period P. The OS then grants priorities to RT I/O threads so that schedulability is guaranteed.

To ensure that a feasible schedule is produce, we are developing admission control strategies for the RT I/O scheduling class. These strategies allow the OS to either guarantee the specified computation time or to refuse to admit the thread. Admission control is useful for both hard real-time and statistical real-time systems.

In addition, we also on solving priority inversion and hidden scheduling[1] problems that arise during protocol processing.

Our strategy for avoiding priority inversion is to have a pool of kernel threads dedicated to protocol processing and then associate these with application level threads.

The kernel-level threads run at the same priority as the application level-threads, which prevents the real-time scheduling hazards mentioned above. In Solaris, both the STREAMS mechanism and the protocol stacks have been multithreaded. Therefore, multiple threads of control can execute within STREAMS modules and multiple connections can proceed simultaneously. We plan to use multiple threads even for uniprocessors to simplify guarantees for periodic data delivery to high-priority applications.

The components describe above, when vertically integrated into the OS, provide hooks and mechanisms for TAO to guarantee Real-Time response to applications.

3.2 Real-time Inter-ORB Protocols in the ORB Core

The ORB Core provides a run-time system that delivers client requests to target objects and returns responses (if any) to clients. TAO's ORB Core uses a Real-time Inter-ORB Protocol (RIOP) as its transport mechanism. TAO's RIOP implementation contains a set of real-time extensions to the General Inter-ORB Protocol (GIOP) [1].

This subsection outlines the existing CORBA interoperability protocols (*i.e.,* GIOP and IIOP) and describes how we extend these protocols to support real-time inter-ORB communication.

3.2.1 Overview of GIOP and IIOP

The CORBA General Inter-ORB Protocol (GIOP) provides end-to-end interoperability between ORBs. GIOP specifies an abstract interface that can be mapped onto transport protocols that meet certain requirements (*e.g.,* connection-oriented, reliable message delivery, untyped bytestream, etc.).

An ORB support 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 transfer syntax called CDR that maps OMG IDL types from the native host format to a bi-canonical format[2] used to send data over the network;

• **GIOP Message Formats:** The GIOP specification defines seven types of 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, as well as describing the constraints on message ordering.

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

3.2.2 Real-time Inter-ORB Protocol (RIOP)

Neither GIOP nor IIOP provide any detailed support for specifying or enforcing the end-to-end QoS requirements of applications. In particular, real-time applications cannot tolerate the latency overhead and jitter of TCP-based GIOP implementations such as IIOP. TCP supports functionality (such as adaptive retransmissions, deferred transmissions, and delayed acknowledgments) that can cause excessive overhead and latency for real-time applications. Likewise, routing protocols like IPv4 lack functionality (such as packet admission policies and rate control) that can lead to excessive congestion and missed deadlines in networks and endsystems.

Therefore, to enhance predictability and performance, TAO supports a Real-time Inter-ORB Protocol (RIOP). RIOP is a mapping of GIOP that allows applications to transfer their QoS parameters end-to-end from clients to target object implementations. TAO's RIOP is designed to transfer QoS information in the `service_context` member of the `GIOP::requestHeader` to ensure compatibility with existing IIOP implementations (which are free to ignore this member).

The TAO RIOP message corresponding to each operation invocation contains attributes that describe the operation's QoS parameters, em e.g., priority, execution period, and communication class. Communication classes supported by TAO include 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) [11].

In addition to transporting application 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 [22]) and can be customized for specific application requirements. For instance, to support applications that do not require complete reliability (*e.g.,* teleconferencing or certain types of imaging), TAO's RIOP mapping can selectively omit transport layer functionality (such as bit-level error detection and retransmissions) and run directly atop ATM or ATM/IP. Likewise, to support avionics applications, TAO's RIOP mapping can transfer the end-to-end priority of an operation with each RIOP message.

[1] Hidden scheduling occurs when the kernel performs work asynchronously without regard to it's priority. STREAMS processing in Solaris is a typical example of this 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.

[2] A bi-canonical format supports both little-endian and big-endian binary data formats.

TAO's RIOP protocol engine is a highly optimized, real-time version of SunSoft IIOP [20] that is integrated with the high-performance I/O subsystem described in Section 3.1. Thus, the ORB Core on the client, server, and any intermediate nodes can collaborate to process RIOP requests in accordance with their QoS class attributes. This design allows clients to indicate the relative priorities of their requests via the ORB QoS Interface described above. ORB endsystems are responsible for enforcing the rate at which periodic IDL operations execute.

To increase portability across OS/network platforms, TAO's RIOP transport implementation is designed as a separate layer in the ORB Core. However, it can be tightly integrated with our underlying Gigabit ATM/IP infrastructure [23, 10, 24, 25, 21].

3.3 QoS Specification via Real-time IDL

Real-time ORBs must allow applications to specify their QoS requirements in order to support end-to-end scheduling and performance guarantees. Several ORB endsystem resources (such as CPU cycles, memory, network connections, and storage devices) are involved in satisfying application QoS requirements. Applications must specify their QoS needs so that the ORB subsystem can guarantee resource availability.

In most hard real-time systems, the amount of computing time required to process client requests must be determined *a priori* so that CPU capacity can be allocated accordingly. In non-distributed, hard real-time systems, CPU capacity is typically the scarcest resource. Furthermore, research in scheduling of real-time systems that considers resources other than CPU capacity relies on on-line scheduling [26]. Therefore, specification of CPU resource requirements is discussed in this section. The QoS mechanism for expressing CPU resource requirements can be readily extended to other shared resources (such as network and bus bandwidth), once the scheduling and analysis capabilities have matured.

In hard real-time systems that are scheduled statically, applications must specify their CPU capacity requirements to an off-line Scheduling Service. This subsection explains how TAO supports QoS specification for the purpose of CPU scheduling for IDL operations that implement real-time tasks. We outline our Real-time IDL (RIDL) schemas (`RT_Task` interface and its `RT_Info struct`), which convey QoS information (*i.e.,* CPU requirements) to the ORB on a per-operation basis.

3.3.1 The RT_Task Interface

The `RT_Task` interface is the mechanism for conveying CPU requirements from tasks performed by application operations to TAO's Scheduling Service, as follows:

```
interface RT_Task
{
  // Time::TimeT is from OMG Time Service.
  typedef Time::TimeT TimeT;   // 100 ns
  typedef Time::TimeT PeriodT; // 100 ns

  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.
  {
    INTERRUPT,       // From highest importance
    IO_SERVICE,
    CRITICAL,        // ...
    HARD_DEADLINE,
    BACKGROUND       // to lowest importance
                     // (deadline not guaranteed
                     //  by Scheduling Service)
  };

  struct RT_Info;   // forward declaration

  struct Dependency_Info
  {
    struct RT_Info dependency;
    unsigned long number_of_calls;
  }

  struct RT_Info
    // The QoS for each operation implementing an
    // application task is described by the
    // following information.
  {
    // = String that uniquely identifies the operation.
    const char *entry_point_;

    // = Execution times.
    TimeT worst_case_execution_time_;
    TimeT typical_execution_time_;

    // = To account for server data caching
    TimeT cached_execution_time_;

    PeriodT period_;
    // 0 means "completely passive", i.e., this
    // operation only executes when called

    Importance importance_;

    TimeT quantum_;
    // for time-slicing (for BACKGROUND tasks only)

    unsigned short number_of_threads;

    sequence < Dependency_Info > task_dependencies_;
    // tasks that we depend upon
  };
};
```

The term "task" is widely used in literature on real-time systems. We felt it was natural to use this term when discussing real-time CORBA scheduling. Thus, in this paper, "task" is synonymous with the implementation of an IDL operation that has real-time properties and execution requirements.

Components that require real-time guarantees can specify the required QoS by inheriting from the `RT_Task` interface. TAO requires applications to specify their QoS information, in this case CPU requirements, by providing the following run-time attributes in an `RT_Info struct`:

- **Worst-case execution time:** The worst-case execution time, *C*, is the maximum execution time that the entry point 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 entry point 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 hard 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, the worst-case execution cost is only incurred once per frame 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 with its own thread of control, then at least one of those threads must execute at least that often. If the operation is passive, i.e., it contains no threads, then the ORB's Object Adapter will invoke this operation once during every period.

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's Object Adapter need not invoke reactive operations periodically, it must account for their execution time when computing the local QoS mapping.

• **Priority:** The task priority is an enumeration value ranging from highest priority (*e.g.,* INTERRUPT) to medium priority (*e.g.,* HARD_DEADLINE), to lowest priority (*e.g.,* BACKGROUND, which runs as a "best effort" task whose deadline not guaranteed).

• **Quantum:** BACKGROUND operations may be time-sliced, *i.e.,* preempted at any time by the OS dispatcher and possibly resumed at a later time. If a time quantum is specified for a BACKGROUND operation, then that is the maximum time that it will be allowed to run before preemption, if there are any other runnable BACKGROUND operations. This time-sliced scheduling provides fair access to the CPU by best-effort, BACKGROUND operations.

• **Called tasks:** This is a vector of other `RT_Info` instances, one for each operation that is called directly by this entry point operation. The called-tasks vectors are used during scheduling analysis to identify threads in the system: each separate called-task chain indicates a thread.

The attributes described above are sufficient for rate monotonic analysis. The RIDL schemas outlined above can be used to specify the run-time execution characteristics of object operations. This information is used by TAO to (1) validate the feasibility of the schedule and (2) allocate ORB endsystem and network resources.

Currently, developers must determine these parameters manually and provide them to TAO's Scheduling Service through its CORBA interface, described in Section 4.3. We are enhancing this process by creating a tool that (1) monitors the execution of applications in example scenarios and (2) automatically extracts the necessary run-time parameters. Likewise, instead of actual execution, simulation results could be used to define `RT_Info` attributes for each operation.

A single `RT_Info` instance is required per entry point in the task, per *mode*. Modes are defined and discussed in Section 4.2.1.

3.4 Real-time Scheduling Service

TAO's real-time Scheduling Service is a CORBA object that has the following off-line and run-time responsibilities:

• **Off-line feasibility scheduling analysis:** TAO's Scheduling Service performs off-line feasibility analysis of all IDL operations that register with the `RT_Info` repository. The goal is to determine whether there are sufficient CPU resources to perform all requested tasks.

• **Thread priority assignment:** During off-line analysis, the Scheduling Service assigns priorities to threads. At run-time, the Scheduling Service provides an interface that allows TAO's Real-time Object Adapter to access these priorities, which are the mechanism for interfacing with the OS-level dispatcher.

• **Coordinate mode changes:** At run-time, the Scheduling Service coordinates mode changes, as described in Section 4.2.1.

Section 4 discusses the Real-time Scheduling Service in detail. Note that our scheduling work in this paper focuses primarily on policies and mechanisms for dispatching CORBA requests correctly in hard real-time avionics systems. However, the architectural features and optimizations we present are applicable to a variety of real-time systems including telecommunications and distributed multimedia systems.

3.5 Real-time Object Adapter

TAO's Object Adapter is responsible for *demultiplexing* and *dispatching* client requests onto object implementations. At run-time, TAO's Object Adapter uses the Scheduling Service to determine how the aggregate processing capacity of ORB endsystems is shared between different application tasks. Therefore, the scheduling and dispatching policies of TAO's Real-time Object Adapter are instrumental to realize end-to-end application QoS requirements.

TAO's Object Adapter can be configured to implement custom mechanisms that dispatch client requests according to desired real-time scheduling policies. TAO's current Object

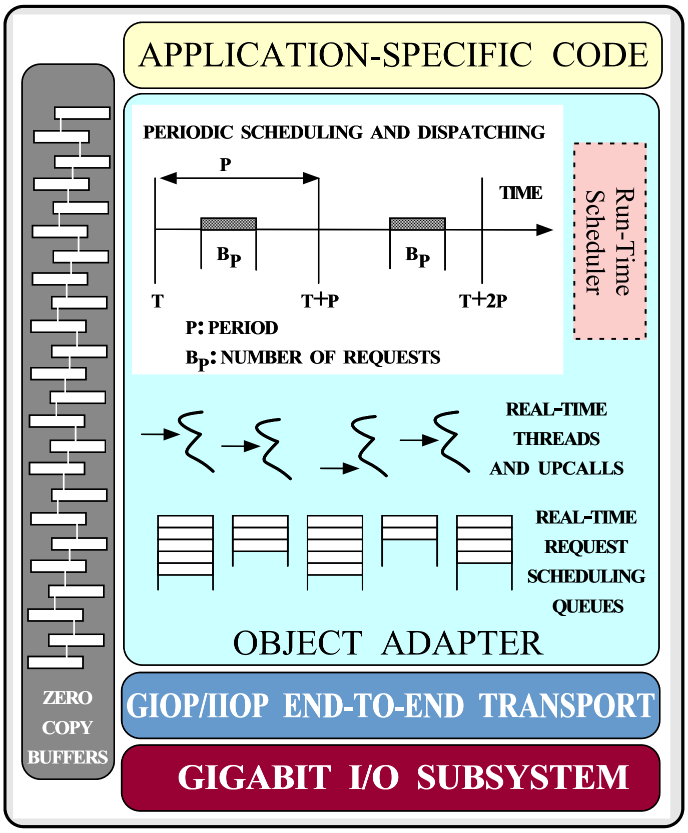

Figure 3: TAO's Real-time Object Adapter

Adapter uses a variant of periodic rate monotonic scheduling with real-time threads and real-time upcalls (RTUs) to provide a guaranteed share of the CPU among application tasks [11, 3].

This subsection describes how TAO's Object Adapter can be configured to dispatch client requests according to various real-time scheduling policies. In addition, we describe how TAO's Object Adapter uses pre-negotiated demultiplexing keys to map client requests directly to object/operation tuples.

3.5.1 Real-time Request Scheduling and Dispatching

TAO's Object Adapter contains an object reference to the run-time scheduler (shown in Figure 3) that dispatches client requests in real-time over uni- and multi-processors in accordance with a system-wide real-time scheduling policy. This run-time scheduler has both off-line and on-line components. The off-line component determines whether a task set can be scheduled, and if so assigns priorities to each of its threads. The on-line component provides a rapid (*i.e.,* $O(1)$ table-lookup) interface to the assigned thread priorities. TAO's Object Adapter dispatches threads to a specific processor according to its *dispatching mechanism.*

Figure 4 illustrates the following dispatching mechanisms supported by TAO's Object Adapter:

- **RTU dispatching (with deferred preemption):** Figure 4(A) shows a single-threaded implementation of TAO's Object Adapter where one real-time OS thread is responsible for queueing and dispatching all requests. This dispatching mechanism model requires consumers to cooperatively preempt themselves when higher priority consumers become runnable. This model of "deferred preemption" is based on a Real-Time Upcall (RTU) concurrency mechanism [11]. A key benefit of the RTU model is its ability to reduce context switching, synchronization, and data movement overhead incurred by preemptive multi-threading implementations. The drawback, of course, is that an RTU model is not automatically preemptive, *i.e.,* it requires cooperation from application operations to perform real-time dispatching.

- **Real-time thread dispatching:** Many OS platforms (*e.g.,* VxWorks and Solaris) now support real-time threading mechanisms that provide worst-case guarantees on thread dispatch time and event response time. Figure 4(B) shows an implementation of TAO's Object Adapter that allocates a real-time thread (or pool of threads) to each priority queue of client requests. By associating appropriate OS priorities to each thread, the Object Adapter can take advantage of kernel support for preemption. For instance, when a thread at priority 0 becomes ready to run, the OS will preempt any lower priority threads (*i.e.,* threads at priority > 0) and allow it to run.

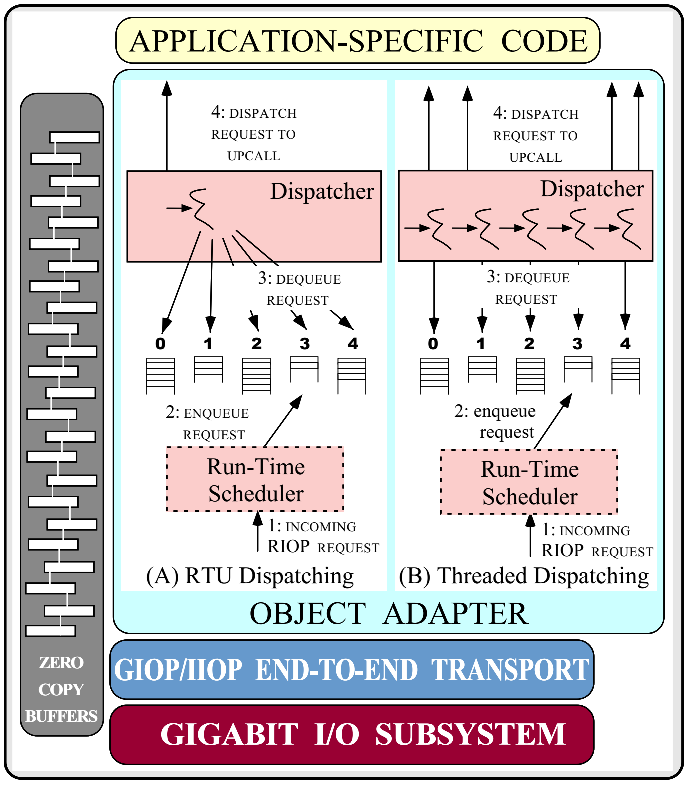

Figure 4: Dispatching Mechanisms for TAO's Real-time Object Adapter

Analysis of a schedule is performed by calculating the utilization and comparing with the utilization bound individually on each processor. We divide processor resources into three classes: (1) *periodic*, (2) *aperiodic with deadlines*, and (3) *best effort tasks*. Rate monotonic scheduling can be used for periodic tasks that require strict real-time service. It can also be used for aperiodic tasks if a worst-case execution interval is assumed. Time-sliced scheduling can be used for best effort tasks, with no guarantee of processor utilization.

TAO's Object Adapter utilizes the Strategy, Bridge, and Abstract Factory patterns [27], which makes it straightforward to configure various real-time scheduling algorithms (such as earliest deadline first) by having an Object Adapter factory create the appropriate dispatching strategy.

3.5.2 De-layered Active Demultiplexing Optimizations

A standard GIOP-compliant client request contains the identity of its remote object implementation and remote operation. The standard remote object implementation is represented by an object key and the remote operation is represented as a string. Conventional ORB endsystems demultiplex client requests to the appropriate operation of the target object implementation using the following steps (shown in Figure 5(A)):

• **Steps 1 and 2:** The OS protocol stack demultiplexes the incoming client request multiple times (*e.g.,* through the data link, network, and transport layers, as well as the user/kernel boundary) to the ORB's Object Adapter;

• **Steps 3 and 4:** The Object Adapter uses the addressing information in the client request to locate the appropriate target object implementation and associated IDL skeleton;

• **Step 5:** The IDL skeleton locates the appropriate operation, demarshals the request buffer into operation parameters, and performs the operation upcall.

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 objects are managed by an ORB. To minimize this overhead, TAO utilizes de-layered active demultiplexing [16, 17] (shown in Figure 5(B)). This approach uses demultiplexing keys that ORB servers assign to clients during connection establishment. These keys map client requests directly to object/operation tuples that perform application-level real-time upcalls.

To further reduce the number of demultiplexing layers, the ATM Port Interface Controller (APIC) describe in Section 3.1 can be programmed to directly dispatch client requests associated with ATM virtual circuits. This strategy reduces demultiplexing latency and supports end-to-end QoS on a per-request or per-object basis.

3.6 Presentation Layer Components and Optimizations

The presentation layer is a major bottleneck in high-performance communication subsystems [28]. This layer transforms typed operation parameters from higher-level representations to lower-level representations (marshaling) and vice versa (demarshaling). In TAO, this transformation process is performed by client-side stubs and server-side skeletons that are generated by a highly-optimizing IDL compiler [29]. This subsection describes the optimizations performed in TAO's presentation layer implementation.

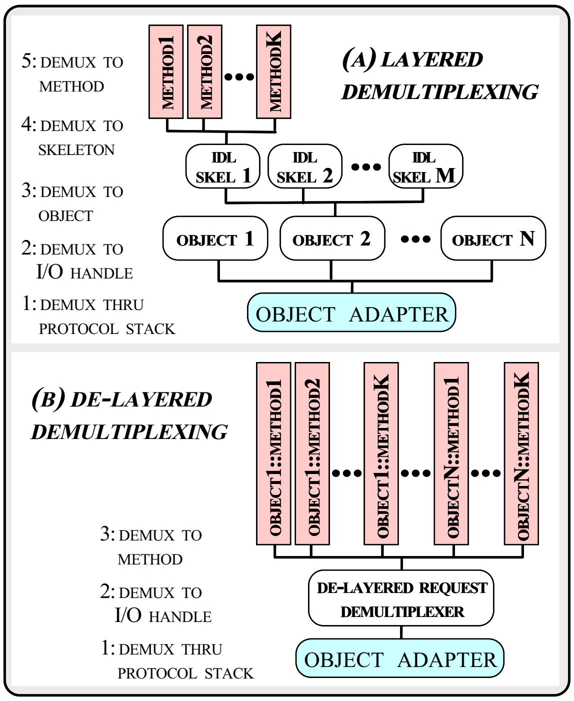

Figure 5: Layered and De-layered CORBA Request Demultiplexing

3.6.1 Presentation Layer Optimizations

The transformation between IDL definitions and the target programming language is automated by TAO's IDL compiler. In addition to reducing the potential for inconsistencies between client stubs and server skeletons, this compiler support innovative automated optimizations. TAO's IDL compiler is designed to generate and configure multiple strategies for marshaling and demarshaling IDL types. For instance, based on measures of a type's run-time usage, TAO can link in either compiled and/or interpreted IDL stubs and skeletons. This flexibility can achieve an optimal tradeoff between interpreted code (which is slow, but compact in size) and compiled code (which is fast, but larger in size) [19].

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 every time. This optimization requires that the real-time ORB perform flow analysis [30, 31] of application code to determine what request fields can be cached.

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

3.6.2 Memory Management Optimizations

Conventional implementations of CORBA suffer from excessive dynamic memory management and data copying overhead. Dynamic memory management is problematic for hard real-time systems because heap fragmentation can yield nonuniform behavior for different message sizes and different workloads. Likewise, excessive data copying throughout an ORB endsystem can significantly lower end-to-end performance [5, 32].

Existing ORBs use dynamic memory management for several purposes. The ORB Core typically allocates a memory buffer for each incoming client request. IIOP demarshaling engines typically allocate memory to hold the decoded request parameters. Finally, IDL dynamically allocate and delete copies of client request parameters before and after an upcall, respectively.

These memory management policies are important in some circumstances (*e.g.,* to protect against corrupting internal CORBA buffers when upcalls are made in threaded applications that modify their input). However, this strategy needlessly increases memory and bus overhead for real-time applications, as well as for streaming applications (such as satellite surveillance and teleconferencing) that consume their input immediately without modifying it.

TAO is designed to minimize and eliminate data copying at multiple points. For instance, the buffer management system described in section 3.1 allows client requests to be sent and received to and from the network without incurring any data copying overhead. Moreover, these buffers can be preallocated and passed between various processing stages in the ORB. In addition, Integrated Layer Processing (ILP) [28] can be used to reduce data movement. Because ILP requires maintaining ordering constraints, we are applying compiler techniques (such as control and data flow analysis) [30, 31]) to determine where ILP can be employed effectively.

Using compiler techniques for presentation layer and memory management functionality allows us to optimize performance without modifying standard OMG IDL and CORBA applications.

4 Real-time Scheduling Service

The TAO RT Scheduling Service is responsible for allocating scarce system resources to meet application needs. For hard real-time systems, the primary function of the Scheduling Service is to guarantee that processing requirements will be met. This section examines the analysis mechanisms and scheduling policies provided by TAO's real-time Scheduling Service. As noted in Section 3.4, it has both off-line and run-time components. Operations are analyzed for schedulability and priorities are assigned off-line; the run-time component provides rapid access to these priority values and coordinates mode changes.

A high-level depiction of these off-line and on-line roles is shown in Figure 6. Starting with the RT_Info instances that the application registers with the Scheduler, the off-line Scheduler constructs chains of called RT_Operations. It identifies threads by looking at the leaves of those call chains. It then assesses schedulability and assigns priorities, outputting the priority tables, per mode, as compilable code.

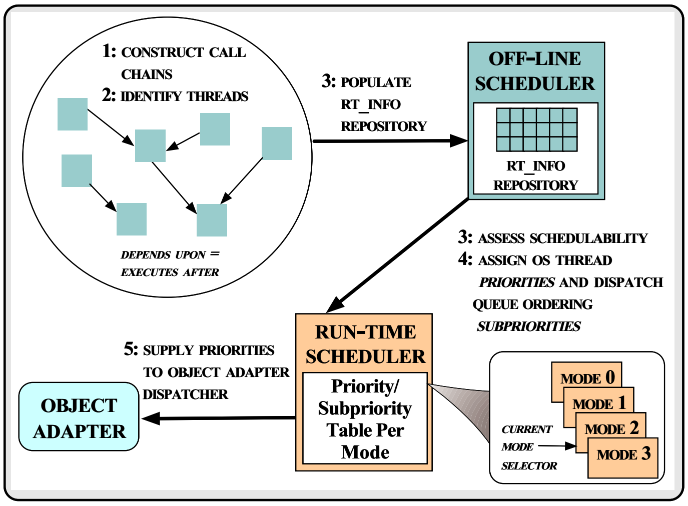

Figure 6: The Scheduler Offline and Online Roles

The Scheduling Service guarantees that all threads 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. This section describes the rate monotonic scheduling implementation used by our current Scheduling Service. We outline the information that the service requires to build and execute a feasible[3] system-wide schedule.

To simplify the presentation, we focus on ORB scheduling for a single CPU. The distributed scheduling problem is not addressed in this paper; it will be addressed in future research.

4.1 Rate Monotonic Scheduling Background

The current implementation of our real-time Scheduling Service uses *rate monotonic scheduling* (RMS) [33, 34]. In RMS, higher priorities are statically assigned to threads with faster rates. This section focuses on static RMS scheduling since our current work is on hard real-time systems.

Other forms of scheduling exist, as well. For instance, *Earliest deadline first* (EDF) scheduling assigns higher pri-

[3] A feasible schedule is one that is schedulable on the available system resources; in other words, it can be verified that none of the tasks will miss their deadlines.

orities to nearer deadlines. However, EDF is commonly used for dynamic scheduling because it permits run-time modifications to rates and priorities. In contrast, static scheduling techniques like RMS require fixed rates and priorities. We currently use only static scheduling, because our avionics application requires assurance of schedulability prior to run-time.

The only analysis technique we have considered thus far is rate monotonic analysis (RMA), which is used to validate the 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 we are not limiting ourselves to RMS.

The primary benefit of using RMA is that it provides a test of whether *schedulability* can or cannot be guaranteed for a system. In the context of hard real-time systems, schedulability is the ability of all tasks in the system to meet their deadlines. We selected rate monotonic scheduling as the first policy for TAO's Scheduling Service because it is straightforward to apply and relatively simple to analyze.

The TAO instantiation of rate monotonic analysis starts with the most strict rules and progressively relaxes restrictions until it reaches a form that can be used. The following restrictions allow simple application of rate monotonic analysis [33, 34]:

- *All tasks are independent*;
- *All tasks are periodic*;
- *There are no interrupts*;
- *Context switching time is negligible*;
- *There is a single task priority level*;
- *There is a single CPU*;
- *Task periods are related harmonically*; and
- *All task 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 task 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 tasks in the program with 1. If the sum of the utilizations is less than or equal to 1, the task set is schedulable; otherwise, it is not.

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

• **Interdependent tasks:** When tasks 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 may only be feasible for hard real-time systems that can be analyzed statically. In such systems, thread activity can effectively be determined prior to run-time.

In statistical real-time systems that have dynamically changing resource requirements, task interdependencies are harder to handle. For instance, there is a potential for *priority inversion* if threads of different priorities can synchronize. By definition, priority inversion occurs when a high-priority task blocks waiting for a low priority task to complete. 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 one of lower priority due to synchronization.

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

• **Interrupts:** Interrupts can be handled readily 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 which may not always occur.

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

• **Multiple task priority levels:** The real-time produces task priorities as outputs. It assigns an operating system-specific priority to each thread in the application, using, for example, RMS. Each task in the thread is then assigned the priority of that thread; for tasks in more than one thread, the highest priority is assigned. RMA can readily 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 [11].

• **Multiple CPUs:** Currently, our RMA analysis assumes TAO's Object Adapter dispatches client requests on 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.

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

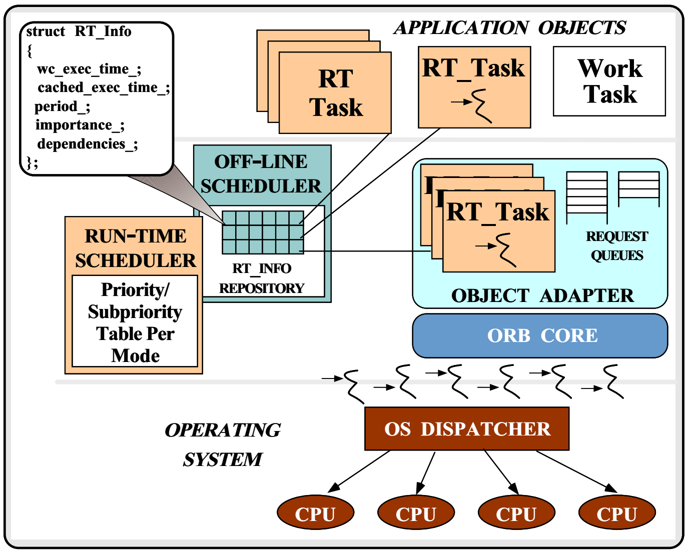

Figure 7: TAO Run-Time Scheduling Participants

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

4.2 Structure and Participants in the TAO Scheduling Service

The TAO Scheduling Service has a run-time component and an off-line component. Participants in the TAO run-time scheduling model include the following, as shown in Figure 7:

• **Work_Task:** A `Work_Task` is a unit of work that encapsulates application-level processing and communication activity. In some real-time environments, a `Work_Task` is called a *module* or *process*, but we avoid these terms because of their overloaded usage in OO contexts.

• **RT_Task:** An `RT_Task` is a `Work_Task` that has timing constraints. Each `RT_Task` is considered to be an "operation" (*i.e.,* a method defined on a target object) 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_Task` instances.

• **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 Win32 thread. All threads are contained within `RT_Task`s. An `RT_Task` containing one or more threads is an "Active Object" [35]. In contrast, an `RT_Task` that contains zero threads is a "Passive Object." Passive Objects only execute in the context of another `RT_Task`, *i.e.,* it must "borrow" the calling task's thread of control to run.

• **OS Dispatcher:** The OS dispatcher uses thread priorities to select the next runnable thread that it will assign to a CPU. It removes a thread from the 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 the lower priority thread. At that point the higher priority, runnable thread can be dispatched onto the CPU.

Our analysis, based on RMA, assumes *fixed priority*, *i.e.,* the operating system does not change the priority of a thread. TAO currently runs on a variety of operating systems (*e.g.,* Solaris [15], Windows NT, and VxWorks) that provide real-time scheduling functionality. Thus, from the point of view of the OS dispatcher and Object Adapter, 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 [36].

• **RT_Info:** An `RT_Info` structure specifies an `RT_Task`'s scheduling characteristics (such as computation time and execution period). `RT_Info`'s are described in more detail in Section 3.3.

• **Run-Time Scheduler:** At run-time, the primary visible vestige of the Scheduling Service is the run-time scheduler, which is used by TAO's real-time Object Adapter. The run-time scheduler returns priorities in response to requests for particular tasks.

Currently, these priorities are statically assigned prior to run-time, and are accessed via an $O(1)$ time table lookup. The run-time scheduler also has a more complicated responsibility: to effect mode changes, as discussed below.

4.2.1 Modes and Mode Changes

A *mode* is an operational regime that is defined by its set of threads and their scheduling-related parameters. For example, power up and air-to-air engagement prior to any weapon firing ay be each represented as modes, or even subdivided into several modes.

A change in membership of the set of threads, or of any thread's scheduling-related parameters, requires a *mode change*. Protocols for implementing mode changes, and their analysis, in priority-driven systems are discussed in [37].

With rate monotonic scheduling, changes in scheduling-related parameters can only affect the schedule via thread priorities. Within a single mode, priorities are fixed. In other words, a new mode must be entered to change a priority. Each mode is analyzed independently for schedulability.

The primary visible run-time component of the Scheduling Service is the mechanism for changing modes. The steps for changing a mode are:

- The run-time scheduler thread is awakened and allocated the CPU;

- The run-time scheduler determines and stores an indication of the new mode;
- The run-time scheduler thread updates its priority tables for the new mode;
- The run-time scheduler thread notifies all application/ infrastructure threads of mode change, either by:
  - Asynchronously notifying the threads (*e.g.* via a signal), or
  - Setting a mode-change preemption flag to indicate to the threads that they must cooperatively preempt themselves (*e.g.*, using an RTU approach); and
- The run-time scheduler thread goes back to sleep.

This design assumes that other threads have some way to asynchronously detect and gracefully react to a mode change. For example, threads could receive a designated signal, perform any necessary cleanup in the signal handler, and immediately return to the run-time scheduler thread.

It is also feasible to provide an alternative way to allow mode changes to happen after all threads have cooperatively preempted themselves (using the mode-change preemption flag mentioned above). However, this approach would add delay prior to the mode change. The tradeoff here is between this delay and the implementation complexity of asynchronous mode-change notification. Other factors could also influence the design, such as a requirement to only effect a mode change on a frame boundary.

4.3 The Off-line Scheduling Service

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

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

- **Thread priority assignment:** If a thread set can be scheduled, then the Scheduling Service assigns a priority to each thread. This is the mechanism that the Scheduling Service uses to convey execution order requirements and constraints to TAO's Object Adapter.

The key types and operations of the IDL interface for TAO Scheduling Service is defined below:

```
#include "RT_Task.idl"

interface Scheduler
{
  typedef unsigned long handle_t;
  typedef unsigned long mode_t;
  typedef short ACE_pri_t;

  enum status_t
  {
    // The schedule() operation has not
    // been called yet.
    NOT_SCHEDULED,
    SUCCEEDED,
    UTILIZATION_BOUND_EXCEEDED,
    INSUFFICIENT_THREAD_PRIORITY_LEVELS,
    CYCLE_IN_DEPENDENCIES,
    VIRTUAL_MEMORY_EXHAUSTED,
    // For use only by register_task.
    TASK_ALREADY_REGISTERED
  };

  typedef sequence <RT_Task::RT_Info> RT_Info_Array;

  // = Registers a task.
  status_t register_task
    (in RT_Info_Array rt_info_array,
     out handle_t handle);
  // If the Task registration succeeds, this
  // function returns SUCCEEDED and sets "handle"
  // to a unique identifier for the task.
  // Otherwise, it returns either the enumeral
  // VIRTUAL_MEMORY_EXHAUSTED or the enumeral
  // TASK_ALREADY_REGISTERED and sets the handle
  // to 0. (a task may only be registered once.)
  // The RT_Info * array is indexed by mode.  If
  // a task does not run in a mode, then its entry
  // in the array for that mode must be 0.

  // = Computes the schedule.
  status_t schedule ();

  // = Access a thread priority.
  ACE_pri_t priority (in mode_t mode,
                      in handle_t handle);
  // Returns the priority that was assigned to
  // the Task that was assigned "handle", for
  // the specified mode.
};
```

Not shown are accessors to system configuration data that the scheduler knows about (such as the number of modes, tasks, and threads in the system, and the current mode identifier). There is also a `destroy` function that developers should call when a program exits. This method allows the scheduler to release its dynamically allocated memory.

In general, this interface need not be viewed by application programmers; the only interface they need to see is the `RT_Info` interface, described in Section 3.3.1. This division of the Scheduler interface into application and privileged sections is shown in Figure 8. The privileged interface is only seen by TAO's Event Service (Section 5), which registers tasks with the Scheduler, by application-level, schedulable operations that do not use the Event Channel, and by the Object Adapter, which accesses the priorities for dispatching.

The remainder of this section overviews the operation of the Scheduling Service: how it assigns thread priorities, when it is invoked, and what it stores in its internal database.

4.3.1 RT_Task Priority Assignments

The Off-line Scheduling Service assigns priorities to each `RT_Task`. Because the current implementation of the Scheduling Service utilizes a rate monotonic scheduling policy, priorities are assigned based on task's rates. For each `RT_Task` in the repository, a priority is assigned based on

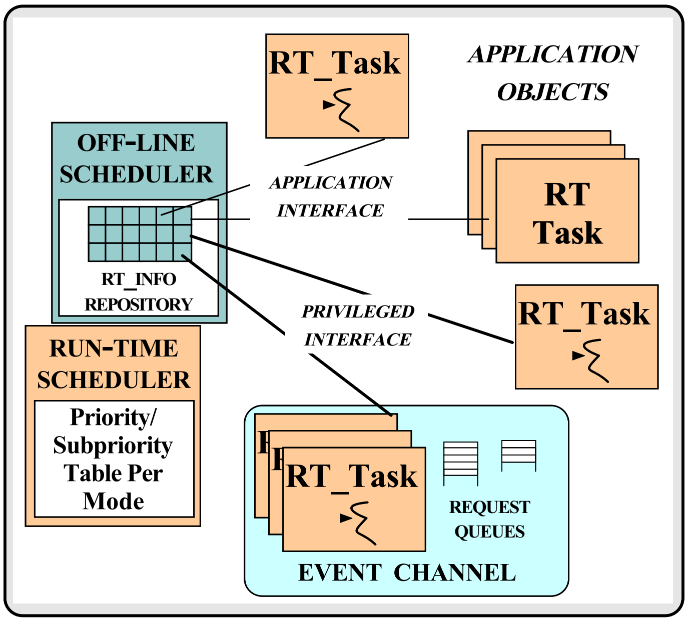

Figure 8: The Two Scheduler Interfaces

the following rules:

- If the task has an non-zero `RT_Info::period`, the value is used to map to a priority. For instance, 100 ms periods may map to priority 0 (the highest), 200 ms periods may map to priority 1, and so on.
- If the task does not have a rate requirement (*i.e.,* its `RT_Info::period` is 0), then its rate requirement must be implied from the `task_dependencies` field in the `RT_Info struct`. The `RT_Info` with the smallest period (*i.e.,* fastest rate) in the `RT_Info::task_dependencies` list will be treated as the task'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 can be queried via a CORBA operation by the run-time scheduler at execution time.

The final responsibility of the Off-line Scheduling Service is to verify the schedulability of a system configuration, *i.e.,* provide a definitive answer to the question "given the current system resources, can the tasks all meet their deadlines?" The Off-line Scheduling Service uses a repository of `RT_Info` structures (shown in Figure 10) to determine the utilization required by each task in the system. By comparing the total required utilization with the known resources, an estimate of schedulability can be calculated.

The Off-line Scheduling Service currently uses the `RT_Info` attributes of application `RT_Tasks` to build the static schedule, as follows:

- *Extract RT_Infos* – Extract all `RT_Info` instances for all the `RT_Tasks` in the system. This information can reside in a Priority Repository and be accessed locally and/or remotely via CORBA requests.
- *Identify real-time threads* – Determining all the real-time threads (*i.e.,* those with hard deadlines, as well as those marked for `BACKGROUND` execution, involves building and traversing task call graphs. These call graphs capture data dependency; if operation A calls operation B, then operation A needs some data that operation B produces. If the two operations execute in the context of a single thread, then operation B must execute before operation A. Therefore, the leaves of the call graphs delineate threads.
- *Determine schedulability and priorities* – Analyze `RT_Info` call graph for all the threads to determine schedulability and assign priorities.
- *Build priority repository* – Emit a table of thread priority assignments, one table per mode. This code is subsequently linked into the application.

These steps are reviewed in the remainder of this section.

4.3.2 Extracting `RT_Info`

The Scheduling Service is a CORBA object that can be accessed by applications during *configuration runs*. To use the Scheduling Service, users must provide `RT_Info` instantiations for each entry point, for each `RT_Task`, per mode, in the system. The steps that the application and infrastructure components must perform for the Scheduler are shown in Figure 9.

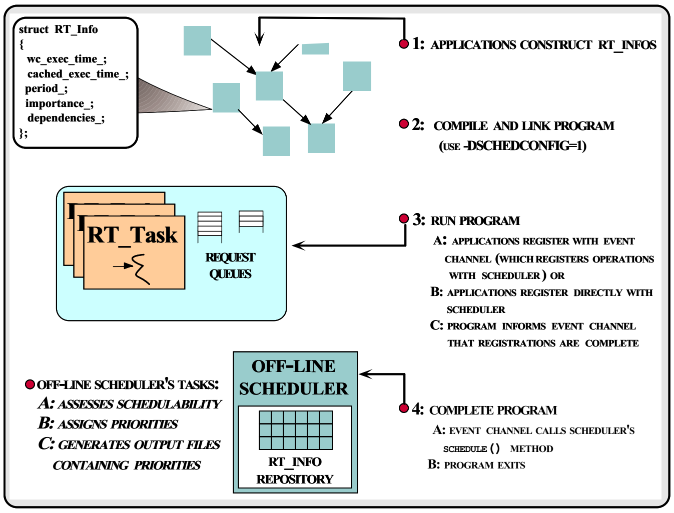

Figure 9: Scheduling Steps During a Configuration Run

The `RT_Info` instantiations are compiled and linked into the main program. With off-line scheduling, they are not needed at run-time.[4]

[4]A future space-saving optimization would be to conditionally compile them in only during configuration runs.

For application objects that register with it, the Event Service determines the dependency information and fills it in the RT_Info, and registers the object with the Scheduler. Schedulable objects that are not Event Service clients must register with the Scheduler directly by filling in all RT_Info information, including dependencies, and registering through the Scheduler register_task interface.

The application notifies the Scheduler after all schedulable tasks have registered, via an Event Service interface provided for this purpose.

The application also notifies the Scheduler when the program is about to exit so that it may release memory that it holds. It is only 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 exits.

For consistency in application and infrastructure code, the Scheduler 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 tasks are registered and that the number of modes and priority values are reasonable.

4.3.3 Identify Real-time Threads and Build Priority Repository

After collecting all of the RT_Info instances, the Scheduling Service uses the following algorithm to identify threads and support its schedulability analysis:

- *Build repository of* RT_Info *instances*:
  - Visit each RT_Info instance; if not already visited, add to repository, and
  - Visit the RT_Info of each called task, depth first, and add a link to the calling task's internal (to the Scheduling Service) called-task vector.
- *Find leaves of called-task trees* – The leaves identify the threads.
- *Traverse each tree* – adding up execution time along the way and identifying the minimum period.
- *Assign priorities using RMS* – *i.e.,* the faster the rate, the higher the priority.

The repository that the Scheduling Service builds is depicted in Figure 10.

The repository includes the RT_Info reference and a vector of RT_Tasks called by each RT_Task. These RT_Task dependencies are depicted by blocks with arrows to the called tasks. These vectors are initialized while traversing the RT_Info instances. Leaves of the called tasks are identified; these correspond to threads. The called-task chains of each thread are traversed to compute the total thread CPU time requirement and identify the minimum execution period.

Passive RT_Tasks (*i.e.,* those without any threads of their own) do not show up in the algorithm as leaves of called-task trees. They may appear further down a called-task chain, in which case their execution time is added to the execution time of the calling thread. "Worst-case" or "cached" execution times can be used depending on whether result caching is enabled and whether the task has been visited already.

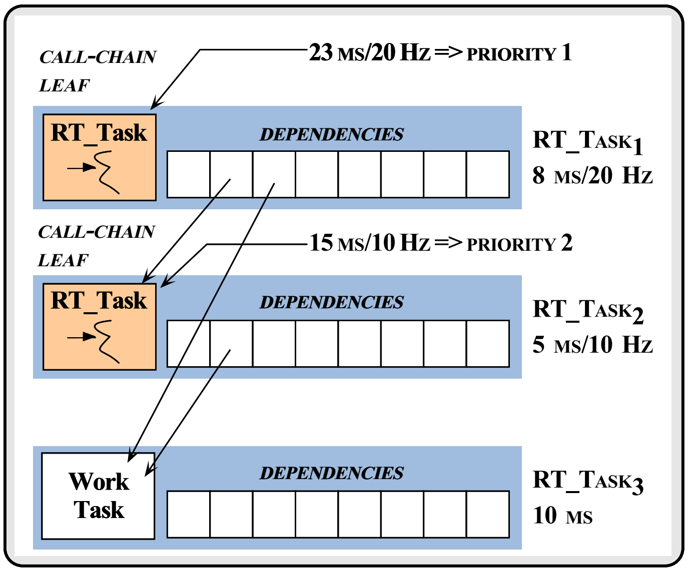

Figure 10: Scheduling Service Internal Repository

This algorithm may appear to complicate the determination of task execution times, because instead of specifying a thread's execution time, an operation's execution time must be specified. However, 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.

The Scheduling Service emits a table, per mode, of thread priority assignments. Every thread is assigned a unique integer identifier. This identifier is used at run-time by TAO's Object Adapter to index into the priority assignment table (which can be accessed in $O(1)$ time since all analysis was performed off-line).

Output from the Scheduling Service is produced in the form of an initialized static search structure 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 Dispatcher Module to access the thread priorities contained in the search structure.

The initial configuration run may contain, at worst, initial estimations of RT_Task 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.

Configuration runs could eventually be used to fill in the called_tasks vectors of the RT_Info instances for each of the RT_Tasks. However, the initial implementation of the Scheduling Service requires that this input be gathered manually.

5 Real-time CORBA Event Service

Many distributed applications exchange asynchronous requests using *event-based* execution models. To support these common use-cases, the CORBA Event Service defines supplier and consumer participants. Suppliers are objects that produce events and consumers are objects that receive events. In addition, the CORBA Event Service defines an Event Channel, which is a mediator [27] that is responsible for propagating events to consumers on behalf of the suppliers.

The OMG Event Service model simplifies application software by allowing decoupled suppliers and consumers, asynchronous event delivery, and distributed group communication [38]. This architecture addresses many of the needs of event-based real-time application domains.

However, the standard CORBA Event Service Specification lacks several features required by real-time applications, including real-time event dispatching and scheduling, periodic event processing, and centralized event filtering and correlations. To resolve the limitations of the CORBA Event Service, we have developed a Real-time Event Service (RT Event Service) [3]. Our RT Event Service augments the CORBA Event Service model with support for source and type-based filtering, event correlations, and real-time dispatching. The architecture of our RT Event Service is shown in Figure 11.

To facilitate real-time scheduling policies (*e.g.,* Rate Monotonic [33]), Event Channels in our RT Event Service can be constructed to have varying levels of support for priority-based event dispatching and preemption. This functionality is implemented using a real-time dispatching mechanism that coordinates with the system-wide real-time scheduling policy described in Section 4. Our RT Event Service is designed to run on OS platforms (such as Windows NT, Solaris, and VxWorks) that provide real-time scheduling guarantees to application threads.

Event-dependent components in real-time systems often carry varying priorities and deadlines. Therefore, our RT Event Service provides predictable event propagation to allow Consumers to receive events within a sufficient time frame to meet their deadlines. Our RT Event Service is currently being deployed in a real-time avionics framework that guarantees deadlines using rate monotonic scheduling and analysis. This framework's scheduling mechanism relies on TAO's ORB endsystem support for priority-based thread preemption and real-time upcalls (RTUs).

Furthermore, many components in real-time systems are characterized by complex event dependencies. For instance, avionics sensor devices execute at fixed rates (*e.g.,* 20 Hz) and depend on data arriving from a periodic I/O source (*e.g.,* a 1553 bus). As a result, our RT Event Channels implement filtering, correlation, and periodic timer mechanisms to guarantee Consumers can process events according to their real-time scheduling requirements.

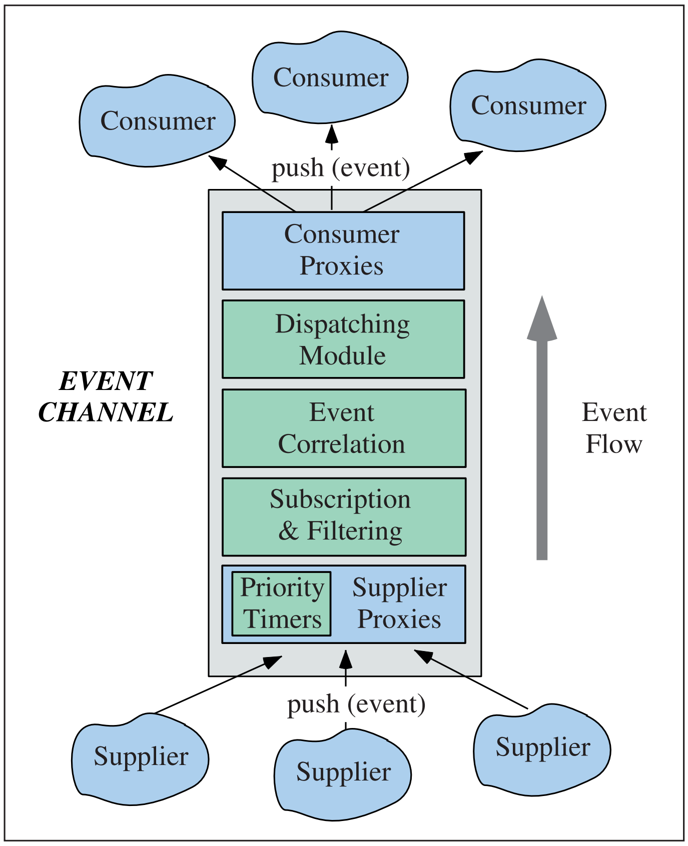

Figure 11: RT Event Service Architecture

6 Concluding Remarks

6.1 Synopsis of TAO's ORB Endsystem Architecture

Currently, there is significant interest in developing high-performance implementations of real-time ORB endsystems [39, 40]. However, meeting these needs requires much more than defining ORB QoS interfaces using OMG IDL – it requires an integrated architecture that delivers end-to-end QoS guarantees at multiple levels of the entire system. The ORB endsystem architecture we describe in this paper addresses this need with the following policies and mechanisms spanning network adapters, operating systems, communication protocols, and CORBA middleware:

- Real-time scheduling of OS and network resources;
- A high-performance ATM Port Interface Controller (APIC);
- Efficient zero-copy buffer management that shares client request buffers across OS protection domains;

- Customized real-time implementations of GIOP-compliant transport protocols;
- A set of Real-time IDL (RIDL) schemas that allows applications to specify QoS attributes for their object's operations using a small number of parameters (such as computation time, execution period, bandwidth and delay requirements);
- A real-time Object Adapter that supports various real-time dispatching mechanisms and de-layered active de-multiplexing;
- An off-line Scheduling Service that determines the priority and scheduling characteristics of client requests with hard real-time deadlines;
- An optimized presentation layer that uses innovative compiler techniques and efficient buffer management schemes to reduce data copying overhead in ORB endsystems;
- Real-time Event Channels that use the RT ORB to support customized scheduling, concurrency, and filtering policies for user-defined events.

6.2 Current Status

TAO's architecture is designed to offer deterministic and statistical guarantees to real-time applications. A prototype of TAO has been completed; it contains the following components (shown in Figure 12):

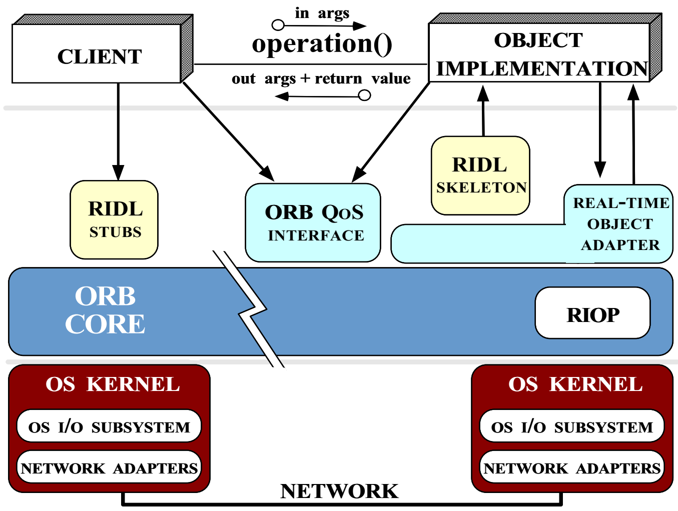

Figure 12: Components in the TAO Real-time ORB Endsystem

- **IDL compiler:** TAO's IDL compiler is based on the Sun IDL front-end (which implements the CORBA 2.0 IDL $\rightarrow$ C++ mapping) [29]. Since TAO is targeted at real-time environments it does not support a DII or DSI C++ language mapping, nor does it provide an interface repository.

- **Inter-ORB Protocols:** TAO contains highly optimized version of the Sun GIOP/IIOP implementation [20], which implements the CORBA 2.0 Internet Inter-ORB Protocol and the real-time RIOP extensions described in Section 3.2.

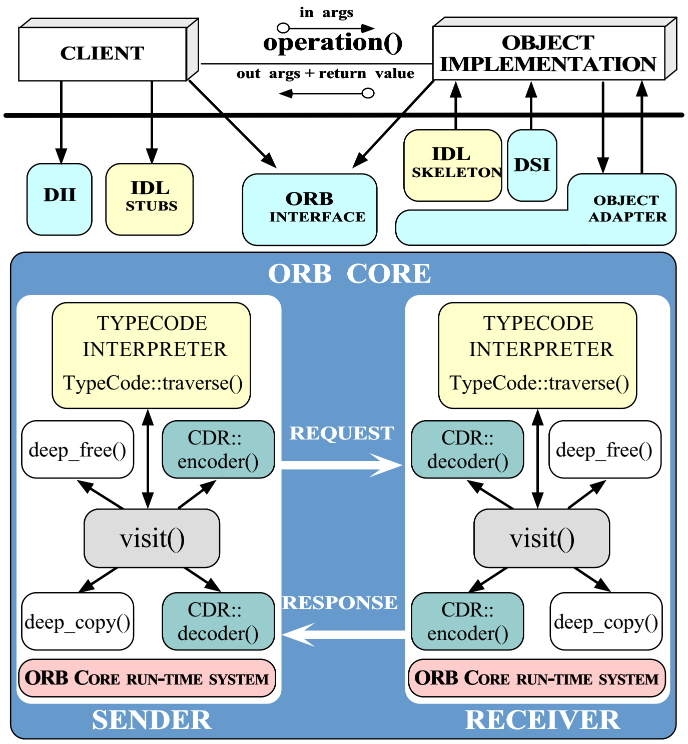

Figure 13: Components in TAO's RIOP Typecode Interpreter

TAO's RIOP implementation is based on a *typecode interpreter* shown in Figure 13. This interpreter marshals and demarshals client requests. The motivation for using an interpreter is to reduce the memory footprint of the marshaling/demarshaling engine so that it resides within the processor cache. The typecode interpreter invokes the `visit` method to marshal or demarshal the data associated with the typecode it is currently interpreting. The `visit` method is a pointer to a method that contains the address of the `encoder`, `decoder`, `deep_copy`, or `deep_free` methods, which perform various tasks during typecode interpretation.

Although TAO does not support a DII language mapping, the underlying protocol marshaling/demarshaling engine supports both the DII and DSI. In fact, the current TAO prototype runs 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic invocation and skeleton interface [20].

- **ACE ORB Core:** The ORB Core provides a run-time system that delivers client requests to target objects and returns responses (if any) to clients. Figure 14 illustrates the components in the client-side and server-side of TAO's ORB Core.

TAO's run-time communication system is based on the

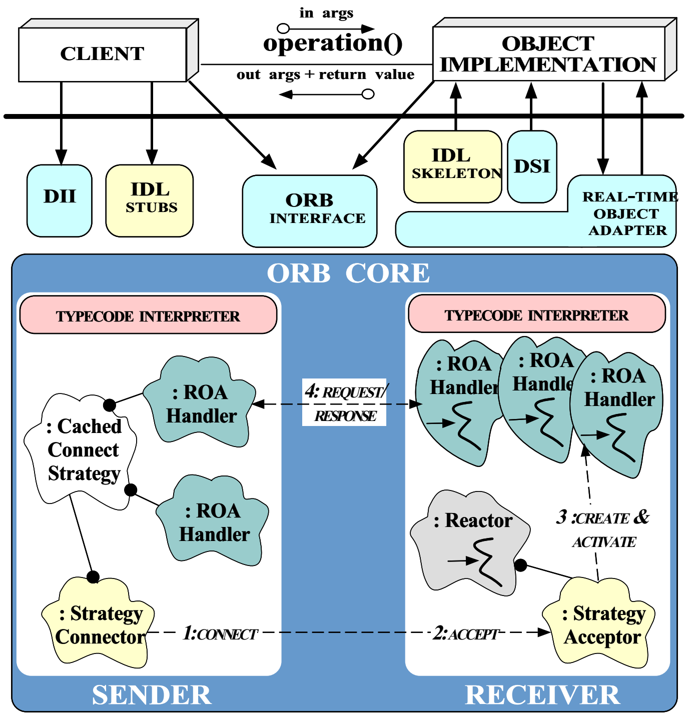

Figure 14: Components in TAO's ORB Core

high-performance, cross-platform ACE [9] components such as Acceptors and Connectors [41], Reactors [9], and Active Objects [35]. ACE is a widely used communication framework developed at Washington University. ACE contains a rich set of high-performance C++ components that implement strategic design patterns [42, 43, 35]. These components automate common communication software tasks such as connection establishment, event demultiplexing and event handler dispatching, message routing, dynamic configuration of services, and flexible concurrency control for parallel network services. ACE has been ported to a variety of real-time OS platforms including VxWorks, Solaris, Win32, and most POSIX 1003.1c implementations.

• **Real-time Object Adapter:** TAO's Object Adapter is designed using various design patterns (such as Strategy [27], Active Object [35], and Half-Sync/Half-Async [43]). These patterns enable TAO's Object Adapter to flexibly implement custom dispatching mechanisms that can process CORBA requests according to different real-time scheduling policies (*e.g.,* rate monotonic and rate monotonic with deferred preemption) [3].

• **Off-line Scheduling Service:** The first implementation of the Off-line Scheduling Service uses rate monotonic scheduling for assigning priorities and rate monotonic analysis for verifying schedulability has been completed. A timeline visualization tool [3] displays request arrival and processing activity, arranged by priority to graphically illustrate the effects of priority inversion and missed deadlines.

The TAO prototype has been used to develop a real-time implementation of the CORBA COS Events Service [3].

Additional information on TAO is available at (http://www.cs.wustl.edu/∼schmidt/TAO.html). Likewise, more information on real-time CORBA is available through the OMG Realtime Special Interest Group (http://www.omg.org/sigs.htm), which provides a forum for defining CORBA standards applicable to isochronous and real-time domains, and from our CORBA WWW page (http://www.cs.wustl.edu/∼schmidt/corba.html).

References

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

[2] 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.

[3] T. Harrison, D. 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. http://www.cs.wustl.edu/∼schmidt/oopsla.html.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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 *IEEE INFOCOM '97*, (Kobe, Japan), IEEE Computer Society Press, April 1997.

[13] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1995.

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

[15] S. Khanna and et. al., "Realtime Scheduling in SunOS5.0," in *Proceedings of the USENIX Winter Conference*, pp. 375–390, USENIX Association, 1992.

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

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

[18] 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.

[19] 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. To be published.

[20] A. Gokhale and D. C. Schmidt, "Optimizing the Performance of the CORBA Internet Inter-ORB Protocol Over ATM," in *Submitted for publication (Washington University Technical Report #WUCS-97-10)*, February 1997.

[21] Z. D. Dittia, J. Jerome R. Cox, and G. M. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," in *IEEE INFOCOM '95*, (Boston, USA), pp. 179–187, IEEE Computer Society Press, April 1995.

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

[23] C. Cranor, "OS Buffer Management System for Continuous Media I/O," tech. rep., Washington University Department of Computer Science, July 1995.

[24] 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.

[25] R. Gopalakrishnan and G. M. Parulkar, "Real-time Upcalls: A Mechanism to Provide Real-time Processing guarantees," Tech. Rep. 95-06, Dept. of Computer Science, Washington University in St. Louis, 1995.

[26] 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.

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

[28] 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.

[29] A. Gokhale, D. C. Schmidt, and S. Moyer, "Tools for Automating the Migration from DCE to CORBA," in *Submitted to ISS 97: World Telecommunications Congress*, (Toronto, Canada), IEEE Communications Society, September 1997.

[30] 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.

[31] 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.

[32] M. B. Abbott and L. L. Peterson, "A language-based approach to protocol implementation," *IEEE Journal of Transactions on Networking*, vol. 1, Feb. 1993.

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

[34] 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.

[35] 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.

[36] 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.

[37] L. Sha, R. Rajkumar, J. Lehoczky, , and K. Ramamritham, "Mode Change Protocols for Priority-Driven Preemptive Scheduling," *J. Real-Time Systems*, vol. 1, pp. 243–264, 1989. John A. Stankovic and Krithi Ramamritham, *Advances in Real-Time Systems*, IEEE Computer Society Press, 1992.

[38] S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," in *Proceedings of the Conference on Object-Oriented Technologies*, (Monterey, CA), USENIX, June 1995.

[39] P. Krupp, A. Schafer, B. Thuraisingham, and V. Wolfe, "On Real-Time Extensions to the Common Object Request Broker Architecture: A Position Paper," in *Proceedings of the OOPSLA Workshop on CORBA*, (Portland, Oregon), October 1994.

[40] Object Management Group, *Realtime CORBA: A White Paper*, 1.0 ed., Dec. 1996.

[41] 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, 1996.

[42] 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.), Reading, MA: Addison-Wesley, 1995.

[43] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[44] 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.

[45] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.

[46] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[47] D. C. Schmidt, "High-Performance Event Filtering for Dynamic Multi-point Applications," in 1^{st} *Workshop on High Performance Protocol Architectures (HIPPARCH)*, (Sophia Antipolis, France), INRIA, December 1994.

[48] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[49] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Flexible and Adaptive Transport System Architecture to Support Lightweight Protocols for Multimedia Applications on High-Speed Networks," in *Proceedings of the* 1^{st} *Symposium on High-Performance Distributed Computing (HPDC-1)*, (Syracuse, New York), pp. 174–186, IEEE, September 1992.

[50] 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.

[51] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, "Language Support for Flexible, Application-Tailored Protocol Configuration," in *Proceedings of the* 18^{th} *Conference on Local Computer Networks*, (Minneapolis, Minnesota), pp. 369–378, IEEE, Sept. 1993.

[52] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[53] D. C. Schmidt and T. Suda, "Experiences with an Object-Oriented Architecture for Developing Extensible Distributed System Management Software," in *Proceedings of the Conference on Global Communications (GLOBECOM)*, (San Francisco, CA), pp. 500–506, IEEE, November/December 1994.

[54] D. C. Schmidt, "A Family of Design Patterns For Flexibly Configuring Network Services in Distributed Systems," in *International Conference on Configurable Distributed Systems*, May 6–8 1996.

[55] 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.

[56] J. O. Coplien and D. C. Schmidt, eds., *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.

Appendix A: Personnel

The PI for the TAO project is Dr. Douglas C. Schmidt, of Washington University. A biography of Dr. Schmidt and the other members of the TAO development group is included in this section.

6.3 Douglas C. Schmidt

Douglas C. Schmidt is an Assistant Professor of Computer Science at Washington University in St. Louis, Missouri. He received his Ph.D. in Computer Science from the University of California, Irvine in 1994. His research focuses on design patterns, implementation, and experimental analysis of object-oriented techniques for developing high-performance distributed object computing systems that run over high-speed ATM networks.

Dr. Schmidt is an internationally recognized expert on distributed object computing and has published widely in ACM, IEEE, IFIP, and USENIX technical conferences and journals on topics including communication software systems, parallel processing for high-performance networking protocols, distributed object computing, and object-oriented design patterns and programming. He served as the program chair for the 1996 USENIX Conference on Object-Oriented Technologies and Systems (COOTS) and the 1996 Pattern Languages of Programming conference.

Dr. Schmidt's research group at Washington University has conducted extensive measurements of the bottlenecks in high-performance communication systems [44, 5, 6, 7]. These measurements reveal why lower-level C and C++ implementations currently outperform object-oriented middleware implementations significantly. The results of these experiments are being used to develop high-performance, adaptive distributed object computing systems and object request brokers (ORBs) [8, 45, 46, 47, 48, 49, 50, 51].

Dr. Schmidt is the chief architect and developer of the ACE object-oriented network programming framework [9, 50]. ACE is a widely used, freely available framework that contains a rich set of high-performance C++ components that implement strategic design patterns [42, 43, 35]. These components automate common communication software tasks such as connection establishment, event demultiplexing and event handler dispatching, message routing, dynamic configuration of services, and flexible concurrency control for parallel network services.

Dr. Schmidt has successfully used ACE in large-scale software systems in research and commercial projects for avionics at McDonnell Douglas [8, 3]; telecommunication systems at Bellcore [29], Ericsson [52, 53, 50], and Motorola [2, 47]; medical imaging systems at Siemens [54, 55] and Kodak [4]. There are currently more than 100 companies and universities using ACE in commercial products and research projects.

Dr. Schmidt's communication software system research on ACE and his current work on the TAO ORB has identified a number of fundamental concurrency and distributed patterns. These patterns represent the core building blocks that are essential to create highly flexible and efficient communication software. For instance, a novel aspect of the ACE framework is its use of design patterns to seamlessly integrate flexible higher-level communication software (such as Object Request Brokers) with efficient lower-level network programming mechanisms (such as sockets, threads, and dynamic linking). Discovering, articulating, and implementing fundamental communication design patterns has enabled Dr. Schmidt to rapidly develop highly efficient applications and object-oriented middleware.

Dr. Schmidt is a co-editor on a book entitled Pattern Languages of Program Design [56]. In addition, he served as guest editor for feature topic issues on Distributed Object Computing for the IEEE Communications Magazine and the USENIX Computing Systems Journal, served as co-guest editor for the Communications of the ACM special issue on Design Patterns and the special issue on Object-Oriented Frameworks, and is an editor of the C++ Report magazine, where he also co-authors a column on distributed object computing with Steve Vinoski, senior architect for US product development of IONA Technologies CORBA Object Request Broker.

Dr. Schmidt has presented keynote addresses and tutorials on reusable design patterns, concurrent object-oriented network programming, and distributed object systems at conferences such as OOPSLA, USENIX, ECOOP, IEEE Local Computer Networks, and ACM Principles on Distributed Computing (PODC). Dr. Schmidt is writing a book for Addison-Wesley with Steve Vinoski on the topic of distributed object programming for a series edited by Brian Kernighan.