TAO: a High-performance 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 hard real-time systems (e.g., avionics) and constrained latency systems (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.

Our scheduling work in this paper focuses primarily on policies and mechanisms for dispatching CORBA requests 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.

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.

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 a periodic IDL operation.

• 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 protocols (GIOP/IIOP) support asynchronous messaging, there is no standard programming language mapping for transmitting ORB requests asynchronously. Likewise, the CORBA 2.0 specification does not require that clients be notified when transport layer flow control occurs. As a result, it is hard to write 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.

This paper describes how we are overcoming these limitations via a high-performance, real-time ORB called TAO.
(The ACE ORB) [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 to run within embedded OS platforms over various real-time interconnects (such as 1553 buses).

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, our work to date focuses on them. Consequently, while the architectural features and optimizations discussed in this paper are applicable to soft real-time systems, (including constrained latency multimedia systems), most of the 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; 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):

- **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 with demultiplexing requests to the target object and dispatching operation upcalls on the object. More importantly, an 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 RT ORB endsystems:

- 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 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, 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 ~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 our 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 our de-layered demultiplexing mechanism and real-time dispatching mechanism that processes client requests 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 [17].

Section 3.1 outlines our 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, [18] 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 our ORB endsystem 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 expected case; eliminating obvious 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 [19].

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, our ORB endsystem 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 demultiplexing 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 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 in real-time; 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 implement end-to-end QoS guarantees, we are developing a high-performance network I/O subsystem. At the heart of this subsystem is a daisy-chained interconnect comprising a number of ATM Port Interconnect Controller (APIC) chips [20]. APIC is designed to sustain an aggregate bidirectional data rate of 2.4 Gbps.

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” of periodic and aperiodic client requests into memory shared among user- and kernel-resident threads [10, 11, 12].

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

When a client invokes an operation, the ORB Core is responsible for delivering the request to the object and returning a response (if any) to the client. 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\(^1\) 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

\(^1\)A bi-canonical format supports both little-endian and big-endian binary data formats.
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, 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 [21]) 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 end-to-end priority rates with each RIOP message.

TAO’s RIOP protocol engine is a highly optimized, real-time version of SunSoft IIOP [19] 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 distinct layer. However, it can be tightly integrated with our underlying Gigabit ATM/IP infrastructure [22, 10, 23, 24, 20].

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 apriori 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 [25]. 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 the Scheduling Service, as follows:

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

   enum Task_Priority
   { // Defines the priorities at which
     INTERRUPT, // From highest priority
     IO_SERVICE,
     CRITICAL, // ...
     HARD_DEADLINE,
     BACKGROUND // to lowest priority
       // (deadline not guaranteed
       // by Scheduling Service)
   };

   struct RT_Info
   // The QoS for each operation implementing an
   // application task is described by the
   // following information.
   {
     // = Execution times.
     // 0 means "completely passive"
     TimeT worst_case_execution_time_;
     TimeT typical_execution_time_;
     TimeT cached_execution_time_;

     PeriodT period_; // tasks that we depend upon

     // for time-slicing (for BACKGROUND tasks only)
     sequence < RT_Info > task_dependencies_;
   };
};
```

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 on a target object.

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 at the entry point operation usually requires. The typical execution time may be useful with some scheduling policies, e.g., soft real-time systems that can relax the conservative worst-case execution time assumption. However, it is not currently used during our hard real-time scheduling.
- **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 Object Adapter’s run-time scheduler, introduced in Section 3.5, 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 timesliced, 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 per 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.

This set of attributes is 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 planning to enhance 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 has the following offline and run-time responsibilities:

- **Off-line feasibility scheduling analysis**: It performs off-line feasibility analysis of IDL operations register with the Scheduling Service’s RT_Info repository to determine whether there are sufficient CPU resources to perform all requested tasks.

- **Thread priority assignment**: During that 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.

### 3.5 Real-time Object Adapter

TAO's Object Adapter is responsible for demultiplexing, scheduling, and dispatching client requests onto object implementations. At run-time, TAO's Object Adapter uses the Scheduling Service to determine how to share the aggregate processing capacity of ORB endsystems 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 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]. For multiprocessor systems, the scheduling mechanism in TAO’s Object Adapter dispatches real-time tasks as shown in Figure 3. This run-time interface to the Scheduling Service admits new real-time client requests arriving at an ORB endsystem consisting of one or more processors and dispatches them to a specific processor.

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 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 3 illustrates the following dispatching mechanisms provided 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.

- **Real-time thread dispatching:** Many OS platforms (e.g., VxWorks, Windows NT, and Solaris) now support real-time threading mechanisms that provide worst-case guarantees on thread dispatch time and event response time. Figure 4(A) 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 and allow it to run.

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, and 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 [26], 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 Demultiplexing Optimizations

A standard GIOP-compliant client request contains the identity of its remote object implementation and remote operation. The 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 at the top of Figure 5):

- **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 demultiplexing [16] (shown at the bottom of Figure 5). 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 APIC 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

The presentation layer is a major bottleneck in high-performance communication subsystems [27]. 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 [28]. This subsection describes the optimizations performed in TAO’s presentation layer implementation.

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.

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 [29, 30] of application code to determine what request fields can be cached.

Although these techniques can significantly reduce mar-
shaping 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 non-uniform behavior for different message sizes and different workloads. Likewise, excessive data copying throughout an ORB endsystem can significantly lower end-to-end performance [5, 31].

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) [27] 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) [29, 30]) 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 primary function of our real-time Scheduling Service for hard real-time systems 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, the TAO Scheduling Service 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.

The Scheduling Service must guarantee 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 implementation of the Scheduling Service. We outline the information that the service requires to build and execute a feasible2 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) [32, 33]. 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 similarly assigns higher priorities to nearer deadlines. However, EDF is commonly used for dynamic scheduling because it permits runtime modifications to rates and priorities. In contrast, static scheduling techniques like RMS require fixed rates and priorities.

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. Rate monotonic scheduling has been selected for TAO 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 [32, 33].

- All tasks are independent;

2A feasible schedule is one that is schedulable on the available system resources; in other words, it can be verified that none of the threads will miss their deadlines.
• 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 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 soft 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 Scheduling Service requires task priority as an input. It assigns an operating system-specific priority to each thread in the application, using, for example, RMS. 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.

• 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 6:

• 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 a “operation” (i.e., a function or method) 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 unit of concurrent execution. A thread corresponds to, e.g., a Solaris or POSIX thread, an Ada task, a VxWorks task, or a Win32 thread. All threads
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. In contrast, time-shared OS schedulers typically age long-running processes by decreasing their priority over time [35]. Thus, from the point of view of the OS dispatcher and Object Adapter, the priority of each thread is constant.

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. The run-time scheduler manages one RT_Info structure for each RT_Task in the system. By using an RT_Task’s RT_Info, the run-time scheduler can be queried for the thread priority of the RT_Task. Currently, the data represented in the RT_Info structures are computed off-line, i.e., priorities are statically assigned prior to run-time. These priorities can be accessed via a simple 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. 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 [36].

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:

1. The run-time scheduler thread is awakened and allocated the CPU;
2. The run-time scheduler determines and stores an indication of the new mode;
3. The run-time scheduler thread updates its priority tables for the new mode;
4. 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
5. 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 interface to the TAO Scheduling Service is defined below:

```c
#include "RT_Task.idl"

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

    enum status_t
    {
        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 VIRTUAL_MEMORY_EXHAUSTED or
    // 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.
};
```

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 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., given the current system resources, can the tasks all meet their deadlines? The Off-line Scheduling Service uses a repository of RT_Info structures 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 all RT_Info instances for all the RT_Tasks in the system. This information can reside in a Priority Repository and be accessed remotely via CORBA requests.
- Identify all real-time threads: those with hard deadlines and those marked for BACKGROUND execution.
- Analyze RT_Info for all threads to determine schedulability and priority assignments.
• Emit a table of thread priority assignments, one table per mode.

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 RT_Info instantiations are conditionally compiled (i.e., protected by a directive such as #ifdef CONFIG_SCHEDULE) and also linked into the main program. Hooks in the RT_Task class cause the RT_Info instances to be registered with the Scheduling Service during configuration runs. The Scheduling Service is thereby able to obtain all of necessary RT_Info input.

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 roots of called-task trees: these identify the threads.

• Traverse each tree, adding up execution time along the way and identifying the minimum period.

• Assign priorities using RMS: the faster the rate, the higher the priority.

The repository that the Scheduling Service builds is depicted in Figure 7.

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. Roots 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 roots 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 7: 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, for example, to accurately consider object-level caching and to provide finer granularity for reuse of RT_Info. 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).

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 production (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. Or 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 [26] 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 [37]. 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 8.

To facilitate real-time scheduling policies (e.g., Rate Monotonic [32]), 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 a system-wide real-time scheduling policy. 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.

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 [38, 39]. 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 demultiplexing;
- 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 end systems;
- 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 9):

- **IDL compiler**: TAO’s IDL compiler is based on the Sun IDL front-end (which implements the CORBA 2.0 IDL → C++ mapping) [28]. 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 [19], which implements the CORBA 2.0 Internet Inter-ORB Protocol, plus the real-time RIOP extensions described in Section 3.2. 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 [19].
- **ACE ORB Core**: TAO’s run-time communication system is based on the high-performance, cross-platform ACE [9] components such as Acceptors and Connectors [40], Reactors [9], and Tasks [41]. 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, 34]. 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 [26], Active Object [34], 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


[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 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, 34]. 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 [28], 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.