Architectural Considerations for Deterministic Real-Time ORB Endsystems and Applications

David L. Levine, Douglas C. Schmidt, and Chris Gill
{levine,schmidt,cdgill}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA *

October 28, 1997

Task 3C/D Deliverable

Abstract

There is increasing demand to extend object-oriented middleware to support applications with stringent quality of service (QoS) requirements. However, conventional object-oriented middleware does not define standard features for specifying or enforcing end-to-end QoS for applications with deterministic or statistical real-time requirements. This paper describes how QoS can be integrated into object-oriented middleware, using the TAO real-time object request broker (ORB). It focuses on the architectural design of the TAO endsystem and on the recommended design of real-time application components that use TAO and CORBA.

Keywords: Real-time CORBA, Object Request Broker, Quality of Service, Object-Oriented Architecture

1 Introduction

1.1 Emerging Trends

Several trends are shaping the future of distributed, real-time system development. First, industry is moving away from *programming* applications from scratch to *integrating* applications using reusable components based on object-oriented (OO) middleware [1]. Second, there is great demand for remote method invocation and messaging technology that simplifies the collaboration of distributed application components [2]. Third, there are increasing efforts to define standard software infrastructures that allow applications to interwork seamlessly throughout heterogeneous environments [3]. Finally, next-generation distributed and real-time applications such as video-on-demand, teleconferencing, and avionics require endsystems that can provide quality of service (QoS) guarantees for latency, bandwidth, and reliability [4].

Existing OO middleware like CORBA [5], DCOM [6], and Java RMI [7] is suitable for request/response applications with best-effort quality of service requirements. However, existing OO middleware is not yet suited for performance-sensitive, distributed real-time applications due to (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) lack of performance optimizations [8].

Several international efforts are currently addressing QoS for OO middleware. The most prominent is the OMG CORBA standardization effort [5]. CORBA allows clients to invoke operations on objects without concern for where the objects reside, what language the objects are written in, what OS/hardware platform they run on, or what communication protocols and networks are used to interconnect distributed objects [3].

Several research efforts [9, 10, 11] are focusing on integrating QoS into the CORBA model. In addition, several OMG groups, notably the Real-Time Special Interest Group and the Telecom Domain Task Force, are actively investigating extensions to CORBA to support distributed real-time applications. This paper addresses several points in the design space for ORB endsystems related to integrating real-time I/O subsystems with CORBA middleware and application components. Our earlier work on CORBA explores other points in the real-time ORB endsystem design space, such as real-time scheduling [9], real-time request demultiplexing [12], and real-time event processing [13].

1.2 The Empirical Imperative

Developing distributed, real-time ORB endsystems is hard. Therefore, it is unrealistic to design these systems using purely analytical techniques. Consequently, the work described in this paper relies on an empirical approach. Our approach

*This work was supported in part by Boeing.

is based on extensive experience developing next-generation avionics [13] and telecommunications [14] systems using OO real-time middleware such as ACE [15] and TAO [9].

ACE is an OO framework that implements core concurrency and distribution patterns [16] for communication software. It provides reusable C++ wrapper facades and framework components that support high-performance, real-time applications and services across a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like VxWorks, Chorus, and LynxOS. TAO is a freely available, highly extensible ORB endsystem written with ACE. TAO is targeted for applications with deterministic and statistical quality of service requirements, as well as best effort requirements.

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

- Identifying the requisite enhancements to standard ORB specifications, such as OMG CORBA, that will enable applications to specify their Quality of Service (QoS) requirements to ORB endsystems [9].
- Empirically determining the features required to build real-time ORB endsystems that can enforce deterministic and statistical end-to-end applications QoS guarantees [9].
- Integrating the strategies for I/O subsystem architectures and optimizations [17] with ORB endsystems to provide end-to-end bandwidth, latency, and reliability guarantees to distributed applications.
- Capturing and documenting the key design patterns [18] necessary to develop, maintain, and extend real-time ORB endsystems.

This paper describes how TAO is being developed to navigate through the design space of real-time ORB endsystem alternatives. We outline key architectural tradeoffs that developers encounter when applying ORBs to open, real-time, distributed applications. Particular attention is paid to the impact of static and dynamic scheduling on the design of application components.

The paper is organized as follows: Section 2 briefly outlines the key features of deterministic real-time, distributed systems; Section 3 gives an overview of CORBA; Section 4 describes the architectural components in TAO's ORB endsystem that support statically scheduled applications with deterministic real-time requirements; Section 5 outlines TAO's current static scheduling approach, motivates dynamic scheduling, and proposes several candidate dynamic approaches; Section 6 evaluates alternative concurrency models for real-time ORB endsystems and application components; Section 7 compares TAO with related work; and Section 8 presents concluding remarks.

2 Architectural Features of Deterministic Real-time, Distributed Systems

Real-time systems and distributed systems are hard to design, implement, debug, validate, and maintain. Moreover, the increasing demand for flexible and extensible open systems has introduced many challenges for developing real-time and distributed systems. Although developing real-time and distributed systems using open architectures is currently beyond the state-of-the-practice, we believe that it is possible to implement such systems by vertically integrating ORB middleware with OS I/O subsystems, communication protocols, and network interfaces [9, 19].

This section reviews the key architectural features of open, real-time, distributed systems. We focus on deterministic applications, such as avionics mission control systems, where meeting periodic processing deadlines is crucial. In addition, we focus on applications that use middleware, such as CORBA, to provide an infrastructure for common application services. We first discuss the high-level characteristics of ORBs, followed by a detailed consideration of the tradeoffs among various features and architectural alternatives.

2.1 Open System Features

Open systems are designed to interoperate with other systems and to be extensible [3]. Interoperability permits decoupled development of systems that often span organizational boundaries. Systems that are extensible can be more easily enhanced and upgraded.

Because we focus primarily on developing reusable middleware for real-time systems, this paper examines *software architectures* for ORB endsystems [9]. An ORB endsystem is more than middleware – it contains the network interfaces, OS I/O subsystem, communication protocols, Object Request Broker (ORB), and ORB services.

Hardware and software are often closely coupled in real-time systems, particularly in embedded systems. A key challenge in developing open systems is to minimize this coupling as much as possible, while still meeting end-to-end application QoS requirements. As described in Sections 4 and 5, the TAO project focuses on integrating configurable software components and flexible real-time scheduling to support these needs.

2.2 Deterministic Real-time System Features

Deterministic real-time systems guarantee that their resources will be available to meet application deadlines. A common, concrete example of real-time guarantees in avionics systems is that sufficient CPU capability must be available to meet all processing deadlines.

Deterministic real-time requirements can be specified by expressing application QoS requirements. Common QoS requirements include CPU time requirements, the periods of periodic tasks or deadlines of aperiodic tasks, importance, and data dependencies. The goal of static scheduling is to demonstrate "off-line" that an application can meet all of its processing deadlines prior to execution.

2.3 Distributed System Features

Distributed systems improve collaboration through connectivity and interworking; performance through parallel processing; reliability and availability through replication; scalability and portability through modularity; extensibility through dynamic configuration and reconfiguration; and cost effectiveness through resource sharing and open systems. However, with these benefits come the following design and implementation challenges:

- **Higher latency and latency variance:** Latency increases for communication across host boundaries, and the variance of latency also increases due to sharing of communication resources.
- **Complex scheduling:** Scheduling becomes more difficult, especially if hard deadlines must be met and communication latencies are variable.
- **Complex mechanisms:** Mechanisms used to achieve the benefits, such as internetworking, parallel processing, replication, and dynamic reconfiguration, are complex, difficult to implement, and their feature interactions are non-trivial.
- **Non-determinism:** Concurrent and distributed processing is often non-deterministic.
- **Partitioning:** Is it hard to partition many real problems to take advantage of distribution.
- **Accidental complexities:** Limitations with tools and techniques, such as low-level network programming APIs like sockets, continuous re-invention of high-level approaches, and a paucity of distributed debugging tools, increase the complexity of developing distributed systems.

The challenges described above make it hard to develop efficient, robust, extensible, portable, and reusable distributed systems.

2.4 Design Constraints on Feature Combinations

Not surprisingly, combining open, real-time, and distributed system features yields various design constraints. Most notably, combining deterministic real-time and distribution requirements necessitates efficient communication channels with bounded, predictable latency. Often, this requirement conflicts with "openness." For instance, most CORBA implementations use TCP/IP as their transport protocol. Although TCP/IP is an open protocol, defined using a widely accepted standard, it is not suitable for deterministic real-time systems due to its potential for jitter and lack of support for message priority.

Feature combination creates additional interactions that complicate scheduling analysis. For example, deterministic real-time systems typically have periodic processing deadlines. Given a static task load and CPU capability, it is possible to determine *a priori* whether that task load is schedulable, *i.e.*, whether all task deadlines will be met. However, combining deterministic real-time requirements with distribution adds another dimension to the analysis: not only are CPU resource requirements and deadlines required for analysis, but the maximum communication latency between each processor is also required.

2.5 Summary

There are few examples of successful open, deterministic real-time, and distributed systems, largely because combining the features and challenges of each type of system is very hard. In particular, feature interaction increases design and analysis complexity and constrains the software architecture of ORB endsystems and application components for these systems.

Our development strategy for COFP is to initially develop an open, real-time system on a single host, using this prototype to address the issues associated with those features first. In this model, distribution was planned for, but not initially implemented. After successful demonstration of a statically scheduled, single processor system [9, 13], we are implementing a distributed solution based on CORBA [5]. In this paper, we investigate the first step toward distribution: evaluation of alternative software architectures for a real-time CORBA ORB endsystem and its application components.

3 Overview of CORBA

CORBA Object Request Brokers (ORBs) allow clients to invoke methods on distributed objects without concern for:

- **Object location:** CORBA objects can be located locally or remotely, without affecting their implementation or use;
- **Programming language:** The languages supported by CORBA include C, C++, Java, Ada95, and Smalltalk, among others.

• **OS platform:** CORBA runs on many OS platforms, including as Win32, UNIX, MVS, and real-time embedded systems like VxWorks, Chorus, and LynxOS.

• **Communication protocols and interconnects:** The communication protocols and interconnects that CORBA can run over include TCP/IP, IPX/SPX, FDDI, ATM, Fast Ethernet, and embedded system backplanes.

• **Hardware:** CORBA shields applications from differences in hardware such as RISC vs. CISC instruction sets.

The components in the CORBA reference model shown in Figure 1 provide this transparency.

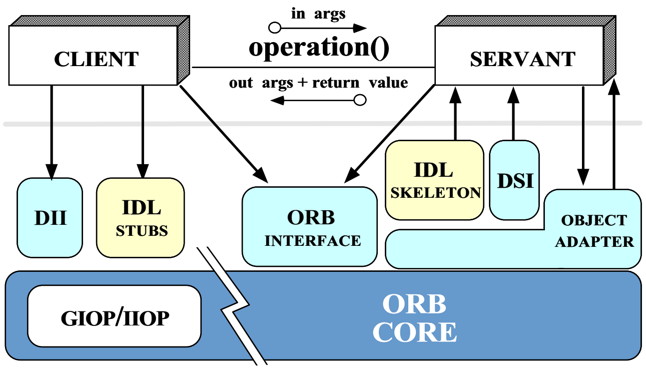

Figure 1: Components in the CORBA Reference Model

These components include the following:

• **Servant:** This defines operations that implement an OMG Interface Definition Language (IDL) interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are typically implemented using one or more objects. A servant is identified by its *object reference*, which uniquely identifies the servant on a server.

• **Client:** This is the program entity that obtains an object reference to a servant and invokes an operation on the servant. The servant can be remote or co-located relative to the client. Ideally, accessing a remote servant should be as simple as calling a method on a local object, *i.e.,* `obj->op(args)`. Figure 1 shows the components that ORBs use to transmit requests transparently from client to servant for remote operation invocations.

• **ORB Core:** When a client invokes an operation, the ORB Core is responsible for delivering the request to the servant and returning a response, if any, to the client. For servers executing remotely, CORBA 2.0-compliant [5] ORB Cores communicate via the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP), which runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked into client and server applications.

• **ORB Interface:** An ORB is a logical entity that may be implemented in various ways, *e.g.*, one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This ORB interface provides standard operations that convert object references to strings and back, as well as create argument lists for requests made through the Dynamic Invocation Interface (DII) described below.

• **OMG IDL Stubs and Skeletons:** IDL stubs and skeletons serve as the "glue" between the client and servants, respectively, and the ORB Core. Stubs provide a strongly-typed, *static invocation interface* (SII) that marshals application data into a common packet-level representation. Conversely, skeletons demarshal the packet-level representation back into typed data that is meaningful to an application. The transformation between OMG IDL definitions and an application programming language like C++ or Java is automated by an IDL compiler. IDL compilers eliminate common sources of network programming errors and increase opportunities for automated compiler optimizations [20].

• **Dynamic Invocation Interface (DII):** The DII allows a client to directly access the underlying request transport mechanisms provided by the ORB Core. The DII is useful when an application has no compile-time knowledge of the interface it is accessing. 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 and receive operations.

• **Dynamic Skeleton Interface (DSI):** The DSI is the server's analogue to the client's DII. The DSI allows an ORB to deliver requests to a servant that has no compile-time knowledge of the IDL interface it is implementing. Clients making requests need not know that the server's implementation uses static skeletons or dynamic skeletons.

• **Object Adapter:** An Object Adapter associates a servant with an ORB, demultiplexes incoming requests to the servant, and dispatches the appropriate operation upcall on that servant. While current CORBA implementations are typically limited to a single Object Adapter per ORB, recent CORBA portability enhancements [21] define the Portable Object Adapter (POA), with support for multiple POAs per ORB.

4 An ORB Endsystem Architecture for Deterministic Real-time Applications

This section motivates the goals of the TAO project and outlines the architectural components in its real-time ORB endsystem. We describe several use-cases that demonstrate TAO's end-to-end and layer-to-layer processing model for statically scheduled real-time client requests. Finally, we review TAO's run-time scheduling mechanism and explain how it has been used to build real-time, event-driven components.

4.1 Limitations of CORBA for Real-time Applications

Our experience using CORBA on telecommunication [14], avionics [13], and medical imaging projects [22] indicates that it is well-suited for conventional request/response applications with best-effort QoS requirements. However, CORBA is not yet suited for performance-sensitive, real-time applications for the following reasons:

• **Lack of QoS specification interfaces:** The CORBA 2.0 standard does not provide interfaces to specify end-to-end QoS requirements. For instance, there is no standard way for clients to indicate the relative priorities of their requests to an ORB. Likewise, there is no interface for clients to inform an ORB how frequently to execute operations that have periodic deadlines. In addition, the CORBA standard does not define interfaces that allow applications to specify their admission control policies. For instance, a video server might prefer to use available network bandwidth to serve a limited number of clients and refuse service to additional clients, rather than admit all clients and provide poor video quality. Conversely, a stock quote service might want to admit a large number of clients and distribute all available bandwidth and processing time equally among them.

• **Lack of QoS enforcement:** Conventional ORBs do not provide end-to-end QoS enforcement, *i.e.*, from application-to-application across a network. For instance, most ORBs use a FIFO strategy for scheduling and dispatching client requests. However, FIFO strategies can yield unbounded priority inversions [23, 24], which occur when a lower priority request blocks the execution of a higher priority request for an indefinite period. In addition, standard ORBs do not control servant execution, *e.g.*, they do not terminate servants that consume excess resources. Thus, existing ORBs largely use *ad hoc* resource allocation. Consequently, a single client can consume all available network bandwidth and a misbehaving servant can monopolize a server's CPU.

• **Lack of real-time programming features:** The CORBA 2.0 specification 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, no standard programming language mapping yet exists for transmitting client 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 [25]. As a result, it is hard to develop portable and efficient real-time applications that behave deterministically when ORB endsystem and network resources are temporarily unavailable.

• **Lack of performance optimizations:** Conventional ORB endsystems incur significant throughput [22] and latency [26] overhead, as well as exhibiting many priority inversions and sources of non-determinism shown in Figure 2. These overheads stem from (1) non-optimized presentation layer conversions and monolithic presentation code that copies data excessively [20] and overflows processor caches [27]; (2) internal message buffering strategies that produce non-uniform behavior for different message sizes [28]; (3) inefficient demultiplexing and dispatching algorithms [12]; (4) long chains of intra-ORB virtual method calls [29]; and (5) lack of integration with underlying real-time OS and network QoS mechanisms [8].

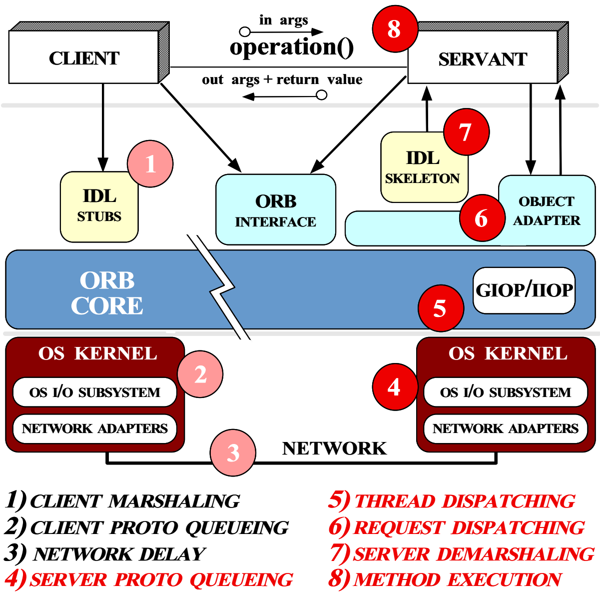

Figure 2: Sources of Latency and Priority Inversion in Conventional ORBs

4.2 Overview of TAO

TAO is an ORB endsystem that contains the network interface, operating system, communication protocol, and CORBA middleware components and features shown in Figure 3. TAO is based on the standard OMG CORBA reference model, with the following enhancements designed to overcome the shortcomings of conventional ORBs for high-performance and real-time applications:

• **Real-time IDL Stubs and Skeletons** In addition to marshaling and demarshaling of operation parameters, TAO's Real-time IDL (RIDL) stubs and skeletons are responsible

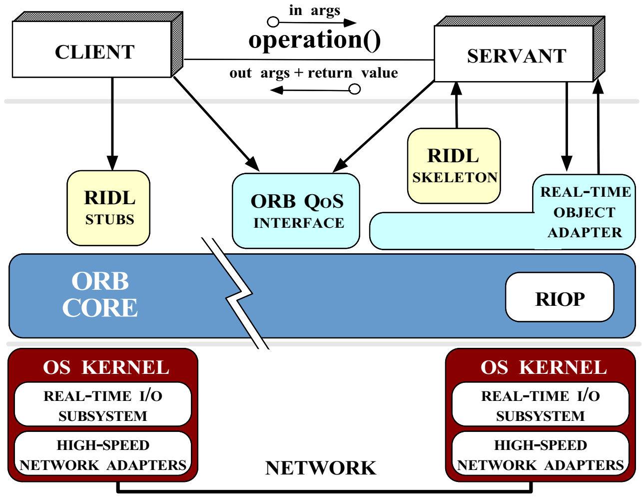

Figure 3: Components in the TAO Real-time ORB

for ensuring that application timing requirements are specified and enforced end-to-end [11].

• **Real-time Object Adapter** In addition to associating servants with the ORB and demultiplexing incoming requests to servants, TAO's Real-time Object Adapter (ROA) dispatches servant operations in accordance with various real-time scheduling strategies such as Rate Monotonic and Earliest Deadline First [9].

• **ORB QoS Interface** Applications use the ORB QoS interface to map real-time processing requirements to ORB endsystem/network resources. Common real-time processing requirements include end-to-end latency bounds and periodic scheduling deadlines. Common ORB endsystem/network resources include CPU, memory, network connections and storage devices.

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

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

4.3 Real-time Support in TAO's ORB Endsystem

The TAO ORB endsystem [9] is designed to support various classes of QoS requirements, including applications with deterministic and statistical real-time requirements. Deterministic real-time applications, such as avionics mission computing systems [13], must meet periodic deadlines. Statistical real-time applications, such as teleconferencing and video-on-demand, can tolerate minor fluctuations in scheduling and reliability guarantees, but nonetheless require QoS guarantees.

To support this range of QoS requirements, TAO's ORB endsystem must be flexible, efficient, predictable, and convenient to program. To meet these requirements, TAO provides end-to-end QoS specification and enforcement mechanisms [9] and demultiplexing optimizations [12] for efficient request dispatching. The following discussion presents an overview of the real-time support in TAO's ORB endsystem and explains how its components can be configured for statically scheduled real-time applications.

4.3.1 Architectural Overview

Figure 4 illustrates the key components in TAO's ORB endsystem. TAO's I/O subsystem runs in the OS kernel and

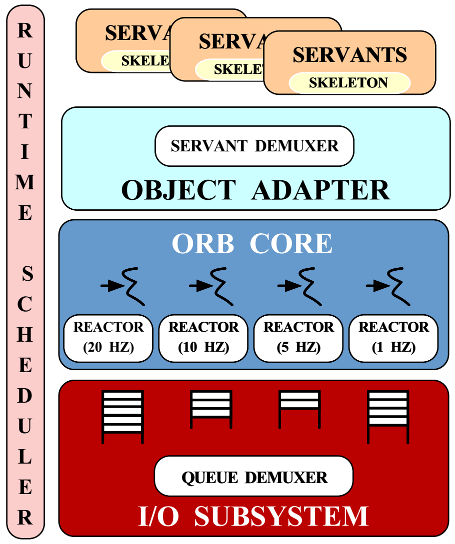

Figure 4: TAO's Real-time ORB Endsystem Architecture

send/receives requests to/from clients across a network or backplane. TAO's ORB Core, Object Adapter, and servants run in user-space. TAO's ORB Core contains a group of real-time threads that process client requests in accordance with their QoS requirements. The Object Adapter demultiplexes the requests using a *servant demuxer*, which associates requests with application-level servant operations. TAO's servant demuxer uses perfect hashing and active demultiplexing [12] to dispatch client requests predictably in $O(1)$ time, regardless of the number of active connections, servants, and operations defined in IDL interfaces.

One or more servants collaborate to implement *application components* or *infrastructure components*. Application components and infrastructure components are reusable building blocks that reside above an ORB. Typically, an infrastructure component represents an entity that is broadly reusable within a particular domain, such as real-time avionics or telecom call processing. In contrast, an application component capture the responsibilities of reusable entities within a particular project, such as the F/A-18, F-15, or AV-8/B aircraft lines. Complete applications can be configured by plugging together application and infrastructure components. In the context of real-time systems, components may or may not be entities that are scheduled wholesale by the ORB endsystem. That is, the boundaries of a component are independent of thread boundaries.

4.3.2 Configuring TAO for Statically Scheduled Real-time Applications

As mentioned earlier, TAO can be configured to support different classes of application QoS requirements. The ORB endsystem configuration shown in Figure 4 is optimized for statically scheduled applications with deterministic, periodic real-time requirements. Figure 5 illustrates an example of this type of application from the domain of avionics mission computing.

Mission computing applications manage sensors and operator displays, navigate the aircraft's course, and control weapon release. The ORB endsystem infrastructure required by these applications must support real-time scheduling and dispatching of periodic processing operations, as well as efficient event filtering and correlation mechanisms [13]. To address these requirements, TAO's ORB Core supports a rate-based concurrency model and a rate-based connection management model.

Rate-based concurrency model: TAO's ORB Core can be configured to allocate a real-time thread for each *rate group*. A rate group assembles all periodic processing operations that occur at a particular rate. Avionics systems commonly execute their tasks in rates groups such as 20 Hz, 10 Hz, 5 Hz, and 1 Hz.

Each rate group thread in TAO's ORB Core contains a `Reactor` that demultiplexes all incoming client requests for its rate group. The `Reactor` used in TAO is an object-oriented implementation of the Reactor pattern [30]. This pattern is used in TAO to demultiplex client requests to GIOP connection handlers. These handlers collaborate with TAO's Object Adapter to dispatch requests to the application-level servant operations.

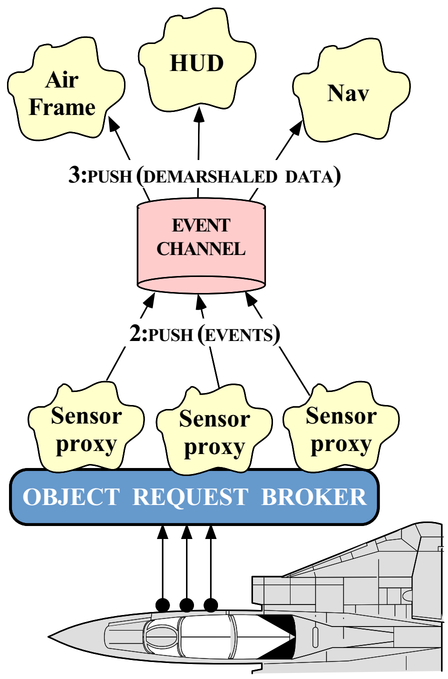

Figure 5: Example Avionics Mission Computing Application

TAO uses a *Reactor, per thread, per rate group* design to minimize the number of threads in the ORB. The primary benefit of this design is its extremely low overhead for context switching and synchronization. In particular, ORB and application component state need only be locked when objects in different rate groups interact. In addition, TAO ORB endsystem concurrency model supports scheduling and analysis techniques that associate priority with rate, such as Rate Monotonic Scheduling (RMS) and Rate Monotonic Analysis (RMA) [31, 32].

Rate-based connection management model: Figure 6 illustrates TAO's rate-based connection management model. Each client thread maintains a `Connector` [33] in thread-

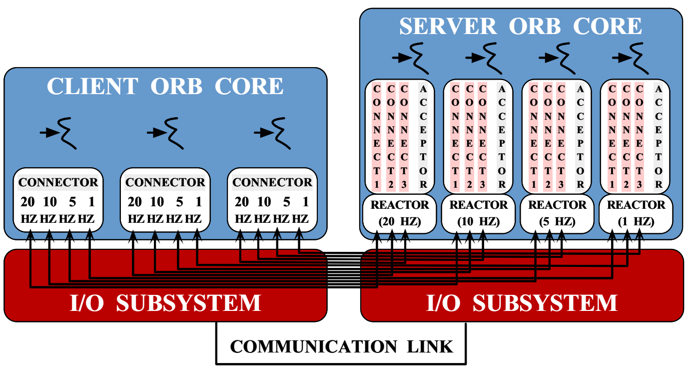

Figure 6: Inter-ORB Connection Topology

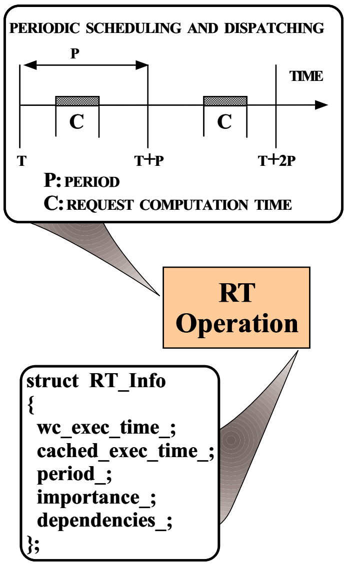

Figure 7: TAO's QoS Specification Model

specific storage. Each `Connector` manages a table of pre-established connections to servers. A separate connection is maintained for each rate. This design enables clients to send client requests through ORB endsystems and across communication links using appropriate end-to-end priorities.

As shown in the server ORB Core in Figure 6, each rate group's `Reactor` is associated with an `Acceptor` [33]. The `Acceptor` is a factory that listens on a particular port number, *e.g.*, 10020, 10010, 10005, 10001, for clients to connect to that rate group. When a client connects, the `Acceptor` creates a new socket queue and GIOP connection handler to service that queue. The I/O subsystem uses the port number contained in arriving requests as a demultiplexing key to associate requests with the appropriate socket queue. This design minimizes priority inversion through the ORB endsystem by directly demultiplexing [17] incoming requests onto the appropriate real-time thread that services the rate group of the target servant.

4.4 Handling Real-time Client Requests in TAO

Real-time applications that use TAO must specify their scheduled resource requirements to TAO's Real-time Scheduling Service [9]. This QoS information is currently provided to TAO on a per-operation basis before program execution. For CPU requirements, the QoS requirements are expressed by `RT_Operations` using the attributes of the `RT_Info` IDL `struct` shown in Figure 7.

An `RT_Operation` is a scheduled operation, *i.e.*, one that has expressed its scheduled resource requirements to TAO using an `RT_Info struct`. The attributes in an `RT_Info` include worst-case execution time, period, importance, and data dependencies. Using scheduling techniques like RMS and analysis approaches like RMA, TAO's Real-Time Scheduling Service determines if there is a feasible schedule based on knowledge of all `RT_Info` data for all the `RT_Operations` in an application. TAO's QoS specification model is described further in [9].

4.4.1 TAO's Outgoing Request Path

Figure 8 shows how TAO dispatches an outgoing client request. This figure traces the steps from the client application, through the proxy, to the I/O subsystem. These steps occur in

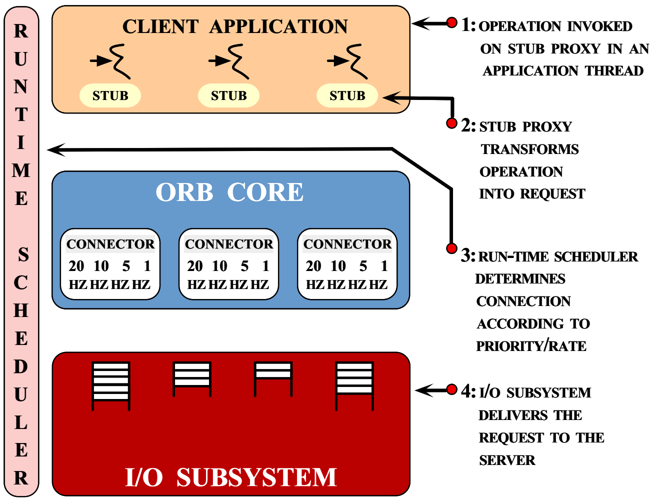

Figure 8: TAO's Outgoing Request Processing

the following order:

Step 1. Operation invocation: An application component running in a real-time thread of control invokes an operation on a object reference. Object references are opaque, immutable "handles" that uniquely identify objects. Each servant must export an object reference before it can be accessed by client applications.

Step 2. Stub transformation: The stub that was automatically generated by the IDL compiler transforms the operation into a request by marshaling the parameters into a common data representation format.

Step 3. Request scheduling and dispatching: The stub consults TAO's Run-Time Scheduler to determine which outgoing connection will send the request over. This determination is based on the rate, *e.g.*, 20 Hz, 10 Hz, *etc.*, at which the request must execute on the server.

Step 4. Request transfer: Finally, the I/O subsystem transfers the request to the server. Requests may be queued in the I/O subsystem according to the configured QoS enforcement policy.

Note that all the state information in TAO, such as the `Connector` maintained by the ORB Core, resides in thread-specific storage. This design eliminates the need for synchronization overhead on the outgoing request path since there is no contention for objects shared by multiple threads. In addition, connections are not multiplexed between threads at different rates. This design improves latency and prevents priority inversion.

4.4.2 TAO's Incoming Request Path

Figure 9 shows how TAO dispatches an incoming client request. This figure traces the steps from receipt of the request by the I/O subsystem to the subsequent upcall to the servant operation designated by the request. These steps occur in the following order:

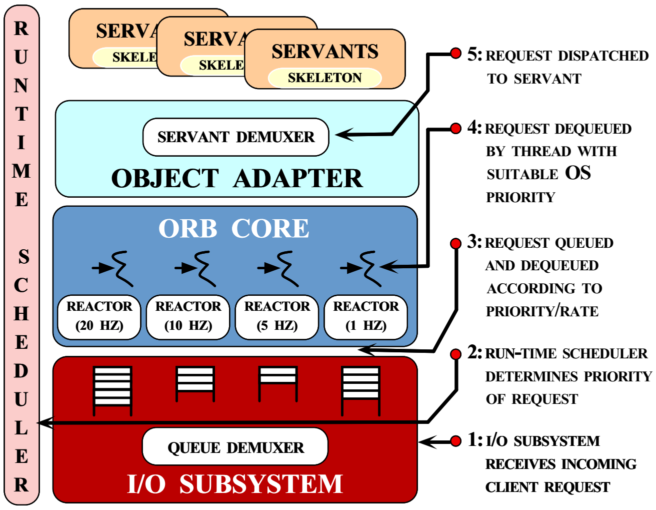

Figure 9: TAO's Incoming Request Processing

Step 1. Request reception: Each real-time thread in TAO's ORB Core manages the socket queues that correspond to the connections for its rate group. Each socket queue is filled by the queue demuxer in the I/O Subsystem. At any given time, the ORB Core thread that executes is the highest priority thread with a non-empty request queue or with processing to be done. For hosts with multiple processors, multiple ORB Core threads can run simultaneously.

Steps 2 & 3. Request scheduling: The I/O Subsystem consults TAO's Run-Time Scheduler to determine in which rate group to queue the request. As described in Section 4.5, TAO's Run-Time Scheduler associates a thread priority with each incoming request to the I/O subsystem in an ORB endsystem. When the request is received, its priority is determined by decoding its header field. If the priority is contained in the request header, it is used directly. For instance, with static scheduling, demultiplexing simply requires a table lookup that locates the priority associated with the destination servant. After finding the priority, the I/O Subsystem enqueues the request directly into the socket queue associated with that priority. With dynamic scheduling, the Run-Time Scheduler must run an algorithm to determine the thread priority when a request arrives.

Step 4. OS thread scheduling and dispatching: After the I/O subsystem demultiplexes a request to its socket queue, the corresponding real-time thread in the ORB Core is dispatched by the OS, according to the priority of the rate group. This thread uses the `Reactor` to dequeue requests from its associated request queue. Because higher rates are associated with higher priority OS threads, this design provides automatic preemption using the native real-time OS dispatcher [24]. Therefore, application components need not implement cooperative preemption. Moreover, data accessed by the ORB endsystem and application components must be locked only if accessible by a servant that runs in more than one rate group.

Step 5. Request dispatching: Finally, the request is passed to the Object Adapter, which demultiplexes the request to the proper servant operation. The servant uses the skeleton generated automatically by the IDL compiler to demarshal the operation parameters and dispatch the upcall.

One or more servants can be registered as event handlers for application components. Therefore, most of the synchronization complexity associated with writing threaded applications does not surface. The exception is for shared data that may be accessed across rate groups. In general, such cross-rate data dependencies should be minimized by application design. However, TAO also supports cross-rate interactions within an ORB endsystem via a loopback pseudo-device.

4.5 TAO's Run-time Scheduling Model

Several TAO components interact with the Run-Time Scheduler shown in Figure 9. The initial TAO prototype only supports off-line, static scheduling. Its Run-Time Scheduler is therefore simply an interface to a table of scheduling information pre-computed off-line. When dynamic scheduling is supported by TAO, the same interface will drive on-line algorithms and retrieve scheduling output interactively.

4.5.1 TAO Scheduling Concepts

The scheduling information used in TAO consists of *priorities* and *subpriorities* that are associated with each servant operation, as follows:

Priority: The priority is the (OS-dependent) thread priority of the highest priority thread than can execute the operation. It is determined based on the maximum rate at which the servant could execute. For example, systems that are scheduled using RMA can compute this information based on knowledge of the computation time, C_o, and period, P_o, for each operation o.

Subpriority: The subpriority orders servant execution within each rate group. This ordering is based on two factors: (1) data dependencies between servants and (2) the relative importance of servants. Data dependencies are expressed directly to TAO's Scheduler via the `RT_Info` QoS specification mechanism described in [9]. In most cases, these dependencies can be determined automatically.

The subpriority of an operation also depends on its *importance*. Importance is an application-specific indication of relative operation importance. It is used by the Scheduler as a "tie-breaker" when other scheduling parameters, *e.g.*, rate and data dependencies, are equal.

4.5.2 Using TAO for Statically Scheduled Real-time Applications

Figure 10 illustrates how priority and subpriority have been used by TAO's Real-time Event Service to order the execution of avionics mission computing operations. TAO's Real-time Event Service is a component that extends the COS Event Service specification [34] to satisfy the QoS needs of real-time applications in domains like avionics, telecommunications, and process control.

TAO's Real-time Event Service simplifies real-time application software by allowing decoupled suppliers and consumers, asynchronous event delivery, real-time event dispatching and scheduling, periodic event processing, and efficient event filtering and correlation mechanisms. In TAO's Real-time Event Service, suppliers generate events, consumers process events received from suppliers, and an *Event Channel* propagates events to consumers on behalf of suppliers in accordance with system-wide real-time scheduling policies.

The Event Channel uses a thread-per-rate concurrency model similar to the one used by TAO's ORB Core described in Section 4.3.2. Each thread manages its request queue, which is filled by the corresponding thread in the ORB Core. The Event Channel's queues are maintained in subpriority order. Each thread dequeues requests from its socket queue, demultiplexes to the proper servant, and issues its `push` upcall. Application components can implement custom `push` operations to process periodic events in a project-specific manner.

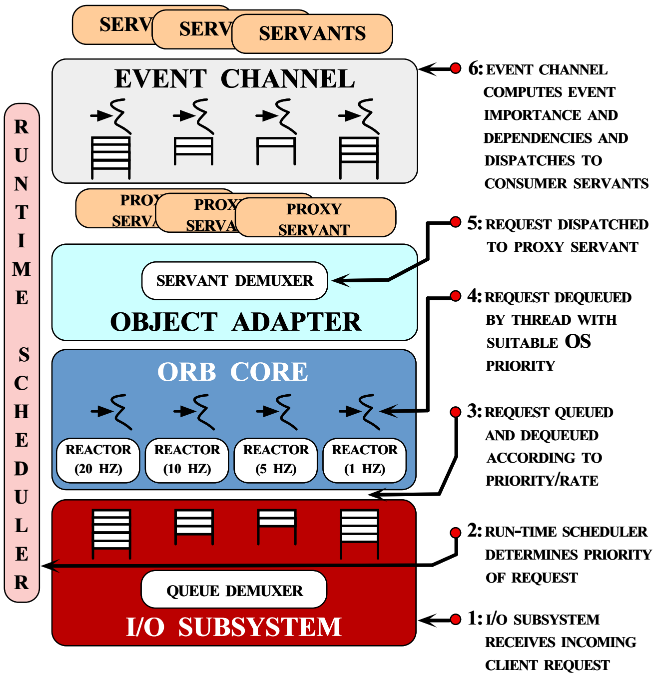

Figure 10: Run-time Scheduling in the TAO Real-time Event Channel

The information necessary to schedule and dispatch client events is computed by TAO's Real-time Scheduling Service based on client QoS and servant resource requirements. TAO's Real-time Scheduling Service interface and its internal operation are described in [9]. TAO's Real-time Event Service is described in [13].

5 ORB Endsystem Scheduling Techniques

The preceding section outlined TAO's real-time scheduling model, focusing on its Run-Time Scheduler. This section explores ORB endsystem scheduling in more depth. First, we review several scheduling approaches, including the off-line, static scheduling initially supported by TAO. We discuss several promising scheduling approaches for open, deterministic real-time, distributed systems, focusing on the impact of the scheduling approach on application components. Finally, we discuss the weaknesses of static scheduling, which motivates the use of dynamic scheduling, as well as a hybrid form of static/dynamic scheduling we are developing for TAO.

5.1 Motivation

A key requirement of deterministic real-time systems is to ensure that all deadlines are met. The purpose of scheduling in a real-time ORB endsystem is to ensure sufficient resources are available to meet the execution-time requirements of application components and ORB endsystem components. In conventional deterministic real-time systems, the CPU is typically the most critical resource. Therefore, most scheduling research focuses on CPU(s). We continue this approach in this paper and concentrate on CPU scheduling.

In general, other resources in a distributed system may be just as critical as the CPU. For example, network and I/O channel bandwidth and endsystem memory may constrain execution ordering. Deadlines may be just as dependent on those resources as on CPU resources. However, the following two factors allow us to focus on the CPU scheduling problem, for now:

1. Current application characteristics: Certain characteristics of an application's environment may reduce the scheduling impact of non-CPU resources. For example, distribution of a real-time system may be over a backplane, such as VME, instead of the local area network of typical distributed systems. Another example is very close control of memory utilization. Deterministic real-time systems typically avoid dynamic memory allocation to prevent its non-deterministic impact on performance. A common alternative is to use a preallocated memory pool. This design has the beneficial side-effect of making total memory requirements explicit (we ignore the issue of memory access contention here).

2. Subsequent generality: We expect that the approaches used for allocating the CPU resource will apply to allocation of other resources, as well. Therefore, when we refer to "scheduling" in this paper, we mean "CPU scheduling" in particular. However, we intend the techniques we discuss to apply to the scheduling of other relevant ORB endsystem resources.

5.2 Real-time Scheduling Techniques

Real-time scheduling techniques can be broadly categorized as *static*, *dynamic*, or a hybrid of the two. With static scheduling, thread priorities are not changed within a scheduling regime and all request priorities are known *a priori*. With dynamic scheduling, the Scheduler may change priorities at run-time.

An example of dynamic scheduling is earliest deadline first (EDF), which adjusts the priorities of all the runnable threads whenever the scheduling algorithm is run. That algorithm assigns higher priorities to runnable tasks with nearer deadlines. The priorities are used directly by the dispatcher to assign the thread(s) with more immediate deadlines to available CPU(s).

In practice, static scheduling is used with *off-line* scheduling, *i.e.*, scheduling execution prior to running the system. Similarly, dynamic scheduling is used with *on-line* scheduling, *i.e.*, executing the scheduling algorithms and assigning priorities at run-time. Therefore, the respective terms of off-line and static scheduling, and on-line and dynamic scheduling, are often used interchangeably. To avoid confusion, we keep these terms distinct in this paper.

5.3 Application Impacts of Static, Off-line Scheduling

Static, off-line scheduling offers several advantages over dynamic, on-line scheduling:

Assurance of schedulability prior to run-time: Off-line scheduling offers assurance prior to run-time that deadlines will not be missed due to insufficient CPU resources. With on-line scheduling, on the other hand, that assurance is not available until run-time. That implies that a resource request may be denied at run-time with dynamic, on-line scheduling. In robust real-time systems, the requester must be prepared to handle this denial of service gracefully.

Lower run-time overhead: Because on-line scheduling requires run-time computation, it imposes more run-time overhead than off-line scheduling. One goal of our real-time ORB endsystem research is to determine if the overhead of sufficiently complex dynamic, on-line scheduling algorithms is prohibitive, relative to the potential gain in utilization.

We anticipate that on-line scheduling can be performed efficiently since certain calculations can usually be performed off-line. In this case, *hybrid* on-line/off-line scheduling approaches may be advantageous. An example of a hybrid approach uses (static) RMS for scheduling hardware interrupt handlers and (dynamic) EDF for scheduling other tasks [31].

Prearranged execution patterns: In some systems, the ordering of task activity is arranged in advance with static scheduling. In the limit, all processing in a statically scheduled system would be precisely "hard-wired," such as with the approach of Xu and Parnas [35]. Application components in such a system could take advantage of such ordering. For instance, a component could assume that the system is in a particular state or it could omit checks for valid input data.

In contrast, dynamic scheduling permits variations in task ordering. For instance, tasks may execute in different orders when run multiple times on a CPU. Since scheduling order is not fixed at design time, application components can not rely on it. However, such reliance on scheduled order of execution at the design phase is anathema to open systems since it tightly couples the design of a component to the design and implementation of other components in the system.

5.4 Limitations of Static Scheduling

While static scheduling provides strong guarantees and has minimum run-time overhead, it has several disadvantages. The following were discussed in [9]:

Distribution cannot be effectively supported: Static scheduling does not support distributed processing well; most scheduling problems on multiple processors are NP-hard [36]. In practice, a specific problem may not be NP-complete or heuristic approaches may be used to develop a static schedule off-line. However, this is tedious and human-intensive task, due to the dearth of existing tool support. Moreover, it is inflexible in the face of configuration changes.

Inefficient handling of non-periodic processing: Static scheduling treats aperiodic processing as if it was periodic, occurring at its maximum possible rate. Resources are allocated to aperiodic operations either directly or through a sporadic server [37] to reduce latency. In typical operation, however, the aperiodic processing may not be required at its maximum possible rate. One example is interrupts, which potentially may occur very frequently, but in many cases do not. Unfortunately with static scheduling, resources must be pessimistically allocated assuming that interrupts occur at the maximum rate. When they do not, utilization is effectively reduced because unused resources cannot be reallocated.

Inflexible handling of invocation-to-invocation variation in resource requirements: Because priorities cannot be (easily[1]) changed at run-time, allocations must be based on worst-case conditions. Thus, if an operation usually requires 5 msec of CPU time, but under certain conditions requires 8 msec, scheduling analysis must assume that 8 msec will be required for every invocation. Again, utilization is effectively penalized because in the usual case, the resource will not be required for 3 msec.

In summary, static scheduling limits the adaptability of systems to changing conditions and changing configurations. In addition, it must compromise resource utilization to guarantee access to resources at run-time. Due to these limitations of static scheduling, we are exploring the use of dynamic scheduling for open, deterministic real-time, distributed systems.

5.5 Towards Dynamic Scheduling

Other forms of scheduling exist beyond static rate monotonic scheduling. For instance, EDF scheduling assigns higher priorities 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 rate monotonic scheduling require fixed rates and priorities. We currently use only static scheduling, because our deterministic avionics applications require assurance of schedulability prior to run-time.

Dynamic scheduling does not suffer from the drawbacks described in Section 5.4. If it can alleviate these drawbacks without too much overhead, then it would be beneficial. However, pure dynamic scheduling does not offer the *a priori* guarantees of static scheduling. For instance, dynamically scheduled systems can behave non-deterministically under heavy loads.

The remainder of this section reviews three promising approaches to dynamic or hybrid static scheduling. These include the algorithm used in the Spring kernel, maximum urgency first (MUF), and two-level hierarchical scheduling.

Spring kernel scheduling algorithm: The Spring kernel was developed at the University of Massachusetts, Amherst. It takes advantage of the *a priori* knowledge of real-time task execution characteristics such as predictable resource utilization patterns and minimum blocking. In contrast, general-purpose operating systems cannot usually rely on expected task execution properties. Furthermore, the goals of general-purpose systems, *e.g.*, maximizing throughput and minimizing average response time, may not be suitable for real-time systems.

Spring handles the scheduling of a newly arrived task as an admission control problem. The new task will be admitted if the set of tasks consisting of the newly arrived task, and all tasks which have already been admitted, can be scheduled. This incremental approach typifies many dynamic scheduling schemes.

Maximum urgency first: Maximum urgency first is the default scheduler for the Chimera real-time operating system (RTOS) [38]. It combines the advantages of rate monotonic, earliest deadline first, and minimum laxity first (MLF) scheduling algorithms [39]. RMS assigns thread priorities based solely on rate of execution. The rate for aperiodic execution must be the assumed worst case. EDF assigns priorities in order of deadlines; the nearer the deadline, the higher the priority. With MLF, priorities are based on more than just relative deadlines, but rather on *laxity*. The laxity for a task is the difference between its deadline, relative to the current time, and its required CPU time. Therefore, MLF considers the execution time of a task, while EDF does not.

RMS is optimal but is limited to systems with a fixed number of tasks with fixed execution requirements [31, 9]. Thus, it is not well suited for dynamic scheduling. EDF supports dynamic scheduling by reassigning priorities whenever the scheduling algorithm is run. However, it does not adequately handle transient overload conditions, when the requested short-term utilization is above 100 percent. MLF is

[1] Priorities can be changed via mode changes, but that is too coarse to capture invocation-to-invocation variations in resource requirements.

similar to EDF in that it reassigns priorities when its scheduling algorithm is run.

To overcome the drawbacks of the individual scheduling algorithms, maximum urgency first (MUF) combines them by assigning both fixed and dynamic priorities. These priorities are summarized in Table 1.

Criticality is a binary (high/low) value that is assigned using RMS. A subset of *critical* tasks are chosen, again based on rate, with worst-case CPU utilization less than 100 percent. This critical set is guaranteed to meet its deadlines. The *dynamic priority* is inversely proportional to laxity. *User priority* is an application-defined priority that is used as a tie-breaker when criticality and dynamic priority are equal. The instantaneous combination of the three priority components – criticality, dynamic priority, and user priority – is called the *urgency*.

MUF has a schedulable bound of 100 percent for the critical set. The critical set is guaranteed to meet all of its deadlines, and it allows the run-time scheduler to detect missed deadlines and notify the offending and affected tasks. Combined with its familiar algorithmic foundation, MUF is a promising scheduling approach for open, deterministic real-time systems.

Two-level hierarchical scheduling: Two-level hierarchical scheduling allows real-time applications to coexist with non-real-time applications in an open environment [40]. As with the Spring kernel scheduling approach, admission control is incremental. A real-time task will only be admitted if it can be scheduled along with all currently admitted real-time tasks. Global scheduling is not required; schedule analysis depends only on the scheduling characteristics of the tasks requesting admission and on the previously calculated scheduling information for the set of currently running tasks.

The scheduling scheme has these objectives:

- Hard deadlines are guaranteed to be met.
- The schedulability of the tasks in each real-time application can be validated in isolation from other applications.
- A simple acceptance test determines whether a new real-time application should be admitted.
- Schedulability is guaranteed for the tasks in admitted real-time applications.
- A certain level of responsiveness is maintained for non-real-time applications.
- It does not rely on fixed allocation or fine-grain time-slicing.

The scheduling system has a two-level hierarchy. Each real-time application has its own *constant utilization server*, and all non-real-time applications share a single constant utilization server. The constant utilization servers are responsible for dispatching for their application(s), and have a fixed size `U` that reflects their application(s)' CPU time requirements. The lower-level OS scheduler coordinates the constant utilization servers. It also decides whether to admit new applications, based on comparison of the sum of the current total `U` with that of the new application, with the processor speed.

The research to date on two-level hierarchical scheduling has focussed on single processor platforms. There are several approaches to extending it to support multiple CPUs. One approach is to treat all remote (interprocessor) activity as non-real-time, thereby minimizing its impact on local real-time applications. Another approach is to create a real-time application, at least for scheduling purposes, to handle interprocessor activity. It would be therefore be allocated resources. For COFP, the latter option is probably preferable. The distributed Event Channel is the natural place for the interprocessor activity.

Other real-time scheduling approaches are reviewed in a "living" web page, which is available at the following URL: `http://www.cs.wustl.edu/~levine/real-time/real-tim`

6 Evaluating Alternative Concurrency Models for Real-time ORB Endsystems and Application Components

Selecting an appropriate concurrency model is one of the most important design decisions for applications built on open, deterministic real-time, distributed ORB endsystems. In this section we compare and contrast alternative concurrency models for applications that execute atop a real-time ORB endsystem. This discussion uses the context and terminology of the TAO Real-Time ORB endsystem [9] and its Real-time Event Service [13]. We categorize the models based on key architectural decisions that must be made early in the design process.

6.1 Relationship to TAO Real-Time Event Service

From an application's perspective, the TAO Real-Time Event Service is the foundation of the COFP execution model. The Real-time Event Service augments the CORBA Event Service model by providing source-based and type-based filtering, event correlations, and real-time dispatching [13]. Its consumer/suppler model provides a familiar execution model.

For performance-critical systems, reduction of blocking overhead is crucial. Therefore, we focus on push-driven models that minimize blocking. Within this model, we categorize consumers and suppliers as passive or active, as shown in Table 2.

Table 1: **MUF Priority Components**

Component	Precedence	Variability	Type	Basis
Criticality	Highest	Static	Binary	Rate
Dynamic Priority	Intermediate	Dynamic	Float	$1/laxity$
User Priority	Lowest	Static	*application defined*	

Table 2: **Event Channel (EC) Participant Categorization**

Event Channel Participant	Passive/Active	Threading	Examples
Consumer	passive	no visible threads	Display; Exception Log
Consumer/Supplier	passive	no visible threads	NAV; Airframe
Supplier/Consumer	active	threaded	Kalman filter
Supplier	active	threaded	EC Reactors; I/O Subsystem

Event Channel participants comprise consumers or suppliers; an object may be both a consumer and a supplier. An object that acts only as a *pure consumer* is passive, and therefore contains no threads that are "visible" to the Event Channel. It may contain threads internally; these threads must be disclosed for scheduling. However, from the Event Channel point of view, these internal threads do not participate in event delivery. Examples of consumers include displays and logging mechanism, *i.e.*, data sinks.

At the opposite end of the spectrum are *pure suppliers*. These must either contain threads of their own or borrow threads, *e.g.*, as an interrupt handler. There typically are very few of these suppliers in an application. Though they are critically important in push-driven systems, because the issue the events that drive the system. Examples include the Event Channel reactor threads, which originate timed events, and application-level objects such as the I/O subsystem, which generates asynchronous events.

The most common participants play the roles of both consumer and supplier. A passive consumer/supplier is a (passive) consumer and a supplier. As a consumer, it waits for events from a different supplier, via the Event Channel. In response to receiving an event, it can push any number of events to the Event Channel, as a supplier. Examples include "interior" application components, such as Navigation processors and the Airframe.

Alternatively, an object that plays both consumer and supplier roles can be active. An active supplier/consumer either contain threads of its own or borrow threads, like a pure supplier. In addition, it can act as a consumer, waiting for events from a different supplier. In response, it can then push events to the channel.

6.2 Scheduling of Consumers and Suppliers

For completeness, Table 2 is extended here to show the relationship of Event Channel participants to TAO's Real-Time Scheduling Service. Consumers and suppliers specify their resource requirements to the Scheduler through the `RT_Info` QoS mechanism, described in [9]. `RT_Info` is an IDL structure that contains resource requirements, such as worst case CPU time, period, importance, and data dependencies.

The Event Channel uses the period for assignment of an operation to one of its dispatching threads. In addition, it fills in the data dependency field before passing the `RT_Info` to the Scheduling Service, via its operation registration interface. Table 3 shows these two `RT_Info` components and how their usage varies with the type of the Event Channel participant.

A pure consumer does not specify a period; its rate of execution depends solely on the rate at which it suppliers push events to it. The Event Channel always fills in the dependencies for a pure consumer, because a pure, useful consumer must have event dependencies if it is to ever run. At the opposite end, a pure supplier must have a rate but has no dependencies. The combined, passive consumer/supplier does not specify a period but has dependencies that are filled in by the Event Channel. Conversely, the combined, active supplier/consumer has both a rate and dependencies.

The final view of the Event Channel participants that we discuss is how they are handled internally by the Scheduler. As part of its scheduling algorithm, the Scheduler constructs directed dependency graphs, analogous to compiler call graphs. Pure consumers are always at the roots of a *dependency chain*, or path through a dependency graph. Pure suppliers are at the leaves. Passive consumer/suppliers and active supplier/consumers are never roots or leaves, but are interior dependency graph nodes. These properties are summarized in the first two columns of Table 4.

Table 3: **The Scheduler External View of Event Channel Participants**

Event Channel Participant	RT_Info	
	Period	Dependencies
Consumer	empty	has dependencies
Consumer/Supplier	empty	has dependencies
Supplier/Consumer	has rate	has dependencies
Supplier	has rate	empty

Table 4: **Other Views of Event Channel Participants**

Event Channel Participant	Scheduler Dependency Chain Location	CORBA designation
Consumer	root	servant
Consumer/Supplier	interior node	client and/or servant
Supplier/Consumer	interior node	client and/or servant
Supplier	leaf	client

This subsection has explored several views of Event Channel consumers and suppliers. To tie them together, Table 4 contains a final column that shows the CORBA designation of each type of Event Channel participant. A supplier acts as a client: it pushes an event, or request, to the channel. A consumer acts as a servant; it accepts requests and operates on them. These roles may seem reversed, but they are a direct consequence of our push architecture.

6.3 Executing Application Components with Event-driven vs. Multi-Threaded Concurrency Models

An important architectural decision is whether the application components in an ORB endsystem should be primarily event-driven or multi-threaded.

The event-driven concurrency model: Application components in an event-driven concurrency model typically share a single thread of control. In response to an event, this thread calls one or more functions, which may depend on the type of the event or system state. Systems built with this concurrency model have "passive" components that borrow the single thread of control as it cycles through various states such as waiting for input events, processing events, and generating output events.

An example of event-driven application components is graphical windowing systems. These applications are often event-driven because that design matches the short-duration, event/response correspondence expected by end users.

The multi-threaded concurrency model: Application components in a multi-threaded concurrency model can execute in multiple threads of control. Threads are designed to perform specific functions, and simply block when necessary, *e.g.*, to receive input, generate output, or synchronize with other threads. Systems built with this concurrency model often have "active" components that possess their own threads of control.

Examples of multi-threaded programs include the client interface to a database engine and Web servers [41]. In the database client example, separate threads may interact with the user and perform updates, to improve response-time and give the user the illusion that updates are instantaneous. Likewise, multi-threaded Web servers allow many clients to retrieve files simultaneously.

There are a number of alternative dimensions in real-time application component concurrency models, as shown in Table 5.

Table 5: **Alternative Dimensions in Real-time Application Component Concurrency Models**

Event-Driven	Multi-Threaded
Reactive	Active
Asynchronous	Synchronous
Push	Pull
Non-blocking	Blocking
One-way	Two-way

The views of reactive versus active, asynchronous versus synchronous, push versus pull, and non-blocking versus blocking follow the event-driven versus multi-threaded distinction, as follows:

- Event-driven systems are typically reactive, whereas multi-threaded systems are naturally active.
- Event-driven systems tend to be asynchronous, especially if handling a single, low-priority event could take a long period of time. The system must then be designed to respond asynchronously to the arrival of higher priority events. In contrast, threading allows the designer to concentrate on individual tasks, letting the OS enforce preemption.
- Push systems are inherently event-driven, while pull systems naturally fit the multi-threaded model.
- Event-driven systems must not block, *e.g.*, waiting on input or remote communication to complete, because no other progress can be made while blocked. With threads, blocking is often the simplest approach. And in some cases the only available solution; for example, in current versions of Java without `select`, waiting on multiple sockets requires the use of multiple threads, which block while waiting for socket activity.
- The CORBA standard only requires that one-way (non-blocking) calls be best-effort. Because they are not reliable, the application is responsible for ensuring delivery. However, some ORBs deliver one-way operations reliably [42]. Furthermore, if distribution is over a backplane, then lower-level transport mechanisms may provide reliability. The ORB core may not take any precautions to avoid overloading servants or the I/O subsystem; in such cases, CORBA services (or the application) must provide flow control.

6.4 Hybrid Approaches

The discussion in this section may appear to assume strict design choices. For example, a given application must either be event-driven or multi-threaded, push or pull, deterministic real-time or best-effort, blocking or non-blocking. However, most complex systems are hybrids; at some level of detail, including hardware internals, *all* reasonably complex systems are hybrids.

Two interesting and common hybrid approaches illustrate the opposite ends of the spectrum of event-driven versus multi-threaded. For instance, in the Half-Sync, Half-Async pattern [43], higher-level tasks, such as application tasks, use a synchronous, *i.e.*, multi-threaded, model for programming simplicity. Lower-level tasks, such as those in the kernel and/or those that manage I/O, use an asynchronous, *i.e.*, interrupt-driven/event-driven, model for execution efficiency. A queueing layer couples the two models. This approach is particularly appropriate for server applications where low-level external events, such as I/O, are most efficiently managed asynchronously, but higher-level functionality is most conveniently processed as synchronously.

An alternative approach uses threads in the lower layers of an application architecture and events in higher layers [44]. Threading in the lower layers effectively supports true concurrency, such as when multiple CPUs or I/O channels are available. The event-driven upper levels insulate application level programmers from the disadvantages of threads, *i.e.*, deadlock and locking of shared data. This approach is particularly appropriate for developing user-interface client applications. Note that these applications must cooperate to assure fairness, avoid starvation, and meet deadlines.

In summary, hybrid systems allow threads to be used where true concurrency is needed. It is especially useful in distributed systems, where remote communication can be managed by threads. This allows concurrently local processing to be managed independently.

6.5 Evaluating the Tradeoffs Between Event-driven and Multi-threaded Concurrency Models for Application Components

In general-purpose systems, there are advantages and disadvantages to developing application components using event-driven vs. multi-threaded concurrency models. In deterministic real-time systems, some of these become more pronounced. For particular applications, they may drive the key architectural decisions.

The most important tradeoffs for open, deterministic real-time, distributed applications include the following:

Minimizing locking: Concurrency control is required to protect most (writable) data that is shared by different threads or in different event handling contexts. This is usually done by *locking* to serialize access to shared data. Locking has significant costs in terms of time, especially because it relies on operating system calls that update kernel structures. Care must also be taken to avoid deadlock, which is not always easy to predict without rigorous analysis, especially if multiple tasks access multiple sets of shared data via separate locks. Furthermore, it is difficult and error-prone to handle unusual situations, such as when an exception is triggered while holding a lock.

Locking is difficult, in general, to manage optimally. Fine grain locking reduces contention, but increases the number of locks, aggravating the difficulties noted in the previous paragraph. Because locking is difficult to implement correctly and optimally, it is best avoided where possible.

Minimizing preemption latency: Preemption latency is the time interval starting when a task with a priority higher than at least one task currently running on a CPU becomes ready to run, and when it starts to run on a CPU. In multi-threaded systems, preemption is typically done via OS facilities, *e.g.*, the OS dispatcher. This provides transparent saving of context, along with a consistent, if not minimal, context switch time.

With a single-threaded system, the application typically provides most of the preemption mechanism via *cooperative multi-tasking*. It is also possible to use asynchronous techniques, such as software interrupts. However, these are more complex and often less efficient than using other OS facilities, *i.e.*, threads, to implement a context switch. Therefore, we have not considered this approach in our applications.

Synchronous preemption does have potential application in some real-time systems. A well-developed implementation is real-time upcalls (RTUs) [45]. Because it uses deferred preemption, preemption latency is sacrificed. However, context switch overhead is greatly reduced because there is less state to save/restore. In addition, preemption only occurs at known points in the application, so that concurrency control does not require locking. Because preemption latency is critical in deterministic real-time applications that we have been working on, we have not considered RTUs.

Threading offers the simplest preemption model from the application perspective. And, its consistent context switch time supports timing analysis, which is necessary for deterministic real-time systems. Furthermore, modern RTOS's are designed to minimize context switching time. For these reasons, we rely on threading for application-level preemption, as well as ORB endsystem preemption.

Maximizing CPU utilization for applications: ORB middleware and the OS must minimize their use of scarce resources, such as CPU time. From the application point of view, any CPU resource that it cannot use is overhead.[2] The choice of an event driven or multi-threaded model can affect this overhead.

With the single-threaded or RTU concurrency models, there is little or no context switching overhead. Therefore, the event-driven approach would seem to be best from the CPU utilization point of view. However, two other factors diminish this advantage. First, as described above, preemption latency is more difficult to control in event-driven systems because application components must cooperate. Second, without implicit context switching, the application must handle interruption of the current event processing in this mechanism when an higher priority event arrives. Context could be explicitly saved, in effect duplicating the activities and overhead of a thread context switch. This negates the apparent advantage of the event-driven approach over threading due to context switching overhead.

[2]However, system developers must be careful not to create such a lightweight infrastructure that application developers must repeatedly reinvent key features in the application layers.

Application design simplicity: Both event-driven and multi-threaded approaches have design restrictions that programmers must obey to built reliable, efficient systems. For example, in an event-driven system with synchronous preemption, the cardinal rule is that application processing must not block. If it does, the entire application blocks, and higher priority events are ignored until the application unblocks and finishes handling its current event.

Programming application components using multiple threads is hard because application developers must ensure that access to shared data is properly serialized in the ORB and in the servants. In addition, the techniques required to robustly terminate servants that execute concurrently in multiple threads are complicated, non-portable, and non-intuitive.

The design simplicity of event-driven and multi-threaded approaches seems to be a matter of taste. After becoming familiar with one of the approaches, it is usually simplest to rely on it when designing other applications.

In summary, the tradeoffs between event-driven and multi-threaded approaches do not indicate a clear preference, in general. Hybrid approaches can take advantage of each one's strengths while minimizing its weaknesses.

7 Related Work

There is a striking similarity between the TAO model and that recommended by Ousterhout [44]. To avoid the difficulties, discussed in Section 6, of threading at the application level, Ousterhout recommends an event-driven model for most applications. But for performance-critical kernel code, Ousterhout recommends that threads may be used in the kernel. If the TAO layers from the Object Adapter and below are viewed as the "kernel" then, because servants are application code, the TAO model corresponds with Ousterhout's recommendation.

TAO readily supports application threads by allowing servants to contain threads and register them with the TAO's Real-time Scheduling Service for system-wide scheduling. Therefore, TAO allows applications to use an event-driven, multi-threaded, or hybrid concurrency models.

The `Reactor`-per-rate-group concurrency model used in TAO is similar to the Real-time Upcall (RTU) concurrency mechanism described in [45]. The primary difference is that RTUs use a single thread to dispatch all the operations in a process. Since the basic RTU model is non-preemptive, this design is prone to longer duration priority inversions than the `Reactor`-per-rate-group used by TAO.

8 Concluding Remarks

Conventional operating systems and ORBs do not provide adequate support for the QoS requirements of distributed, real-time applications. Meeting these needs requires an integrated ORB endsystem architecture that delivers end-to-end QoS guarantees at multiple levels of the entire system. The TAO ORB endsystem architecture described in this paper addresses this need by combining a real-time I/O subsystem with a carefully design ORB Core and Object Adapter. The source code for ACE and TAO is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

Due to the limitations of static scheduling, it is clear that at least some component of dynamic scheduling is required for open, deterministic real-time, distributed systems. We have explored the architectural features of applications, with close attention to alternatives that facilitate scheduling while maintaining good performance.

Acknowledgements

We gratefully acknowledge the support and direction of the Boeing Principal Investigator, Bryan Doerr. In addition, we thank Tim Harrison for productive discussions that helped shape our thinking on the architectural features of open, deterministic real-time, distributed systems.

References

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

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

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

[4] R. Gopalakrishnan and G. Parulkar, "Quality of Service Support for Protocol Processing Within Endsystems," in *High-Speed Networking for Multimedia Applications* (W. Effelsberg, *et. al.*, ed.), Kluwer Academic Publishers, 1995.

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

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

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

[8] 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, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, 1997, to appear.

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

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

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

[13] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[14] 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.

[15] 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.

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

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

[18] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," *Communications of the ACM, to appear*, Dec. 1997.

[19] D. C. Schmidt, R. Bector, D. Levine, S. Mungee, and G. Parulark, "TAO: a Middleware Framework for Real-time ORB Endsystems," in *Submitted to the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[20] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

[21] Object Management Group, *Specification of the Portable Object Adapter (POA)*, OMG Document orbos/97-05-15 ed., June 1997.

[22] 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.

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

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

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

[32] 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.

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

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

[35] J. Xu and D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, vol. 16, pp. 360–369, Mar. 1990.

[36] J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, vol. 28, pp. 16–25, June 1995.

[37] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced Aperiodic Scheduling in Hard Real-Time Environments," in *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, 1987.

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

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

[40] Z. Deng and J. W.-S. Liu, "Scheduling Real-Time Applications in an Open Environment," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, Dec. 1997.

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

[42] S. Baker, *CORBA Distributed Objects using Orbix*. Addison Wesley Longman, 1997.

[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] J. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)," in *USENIX Winter Technical Conference*, (San Diego, CA), USENIX, Jan. 1996.

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