Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA

Carlos O'Ryan and Douglas C. Schmidt
{coryan,schmidt}@uci.edu
Electrical & Computer Engineering Dept.
University of California, Irvine
Irvine, CA 92697, USA

Fred Kuhns, Marina Spivak, Jeff Parsons,
Irfan Pyarali, and David L. Levine
{fredk,marina,parsons,irfan,levine}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

This paper appeared in the Wiley & Sons Concurrency and Computing: Practice and Experience Journal, Volume 13, Number 2, 2001.

Subject Areas: Real-time CORBA; Patterns and Frameworks; Distributed and Real-Time Middleware

Abstract

To be an effective platform for performance-sensitive real-time systems, commodity-off-the-shelf (COTS) distributed object computing (DOC) middleware must support application quality of service (QoS) requirements end-to-end. However, conventional COTS DOC middleware does not provide this support, which makes it unsuited for applications with stringent latency, determinism, and priority preservation requirements. It is essential, therefore, to develop standards-based, COTS DOC middleware that permits the specification, allocation, and enforcement of application QoS requirements end-to-end.

The Real-time CORBA and Messaging specifications in the CORBA 2.4 standard are important steps towards defining standards-based, COTS DOC middleware that can deliver end-to-end QoS support at multiple levels in distributed and embedded real-time systems. These specifications still lack sufficient detail, however, to portably configure and control processor, communication, and memory resources for applications with stringent QoS requirements.

This paper provides four contributions to research on real-time DOC middleware. First, we illustrate how the CORBA 2.4 Real-time and Messaging specifications provide a starting point to address the needs of an important class of applications with stringent real-time requirements. Second, we illustrate how the CORBA 2.4 specifications are not sufficient to solve all the issues within this application domain. Third, we describe how we have implemented portions of these specifications, as well as several enhancements, using TAO, which is our open-source real-time CORBA ORB. Finally, we evaluate the performance of TAO empirically to illustrate how its features address the QoS requirements for certain classes of real-time applications.

*This work was funded in part by AFOSR grant F49620-00-1-0330, Boeing, NSF grant NCR-9628218, DARPA contract 9701516, Motorola, Nortel, SAIC, Siemens, and Sprint.

1 Introduction

Challenges for next-generation real-time systems: Due to the need to handle stringent constraints on efficiency, predictability, memory footprint, and weight/power consumption, software techniques used to develop real-time systems have historically lagged behind those used to develop mainstream desktop and server software. As a result, real-time software applications are difficult to evolve and maintain. Moreover, they are often so specialized that it is not cost effective to adapt them to leverage new technology innovations or to meet new market opportunities.

To exacerbate matters, a growing class of distributed real-time systems require end-to-end support for various quality of service (QoS) aspects, such as latency, jitter, and throughput. These applications include the control and management of telecommunication systems, commercial and military aerospace systems, and streaming audio/video over the Internet. In addition to requiring support for stringent QoS requirements, these types of systems are often targeted for markets where deregulation, global competition, and/or R&D budget constraints necessitate increased software productivity.

Requirements for increased software productivity motivate the use of distributed object computing (DOC) *middleware* [1], such as CORBA [2] and Java RMI [3]. DOC middleware resides between applications and the underlying operating systems, protocol stacks, and hardware in complex distributed and embedded real-time systems. The *technical goal* of DOC middleware is to simplify software development by shielding applications from component location, programming language, OS platform, communication protocols and interconnects, and hardware dependencies [4]. The *business goal* of DOC middleware is to decrease the cycle-time and effort required to develop real-time applications and services.

In theory, middleware can simplify the creation, composition, and configuration of real-time applications without incurring significant time and space overhead. In practice, however, technical challenges have impeded the development and deployment of efficient, predictable, and scalable middleware for real-time systems. In particular, commodity-off-the-shelf (COTS) DOC middleware generally lacks (1) support for QoS specification and enforcement, (2) integration with high-speed networking technology, and (3) efficiency, predictability, and scalability optimizations [5]. These omissions have limited the rate at which performance-sensitive applications, such as teleconferencing and avionics mission computing, have been able to leverage advances in DOC middleware.

Candidate solution $\rightarrow$ CORBA: First-generation DOC middleware was not targeted for high-performance and real-time systems. Thus, it was not appropriate for systems with stringent deterministic and statistical real-time QoS requirements [5]. Over the past two years, however, the use of CORBA-based DOC middleware for real-time applications has increased significantly in aerospace [6], telecommunications [7], medical systems [8], and distributed interactive simulation [9] domains. The increased adoption of CORBA stems from the following factors:

1. *The maturation of patterns* – In recent years, a substantial amount of R&D effort has focused on patterns [10, 11]. For instance, research conducted as part of the DARPA Quorum project [12, 7, 5, 13] has identified key patterns [14] and optimization principles [15] for high-performance and real-time systems.
2. *The maturation of frameworks* – Recent progress in patterns R&D has enabled the creation of higher-quality frameworks [16], such as ACE [17], that support the development of QoS-enabled DOC middleware and applications.
3. *The maturation of standards* – During the past decade, the OMG's suite of CORBA standards has matured considerably, particularly the Real-time [18] and Messaging [19] specifications that define components and QoS features for high-performance and real-time systems.
4. *The maturation of COTS CORBA products* – An increasing number of COTS ORBs [20] are applying patterns and frameworks to implement the CORBA Real-time [5] and Messaging [21] specifications.

The vehicle for our research on DOC middleware for high-performance and real-time applications is TAO [5]. TAO is an open-source CORBA-compliant ORB designed to support applications with stringent end-to-end QoS requirements. In our prior work on TAO, we have shown that it is possible to achieve high efficiency, predictability, and scalability in ORB middleware by applying appropriate concurrency [14], connection [22] and demultiplexing [15] patterns [1].

Our earlier work, however, has not addressed techniques for balancing competing real-time application requirements for latency and throughput. Moreover, the OMG has recently approved the Real-time [18] and Messaging [19] specifications, which give application developers greater control over end-to-end priority preservation and ORB predictability. Therefore, in this paper, we evaluate these specifications to illustrate the extent to which they do and do not satisfy the requirements of an important class of real-time applications. For situations where CORBA 2.4 is under-specified, we demonstrate how the specification can be enhanced to allow greater application control and portability.

Paper organization: The remainder of this paper is organized as follows: Section 2 presents an overview of the OMG CORBA specifications relevant to this paper; Section 3 outlines key design forces affecting an important class of real-time applications and describes an avionics mission computing [6] application that is representative of these types of applications; Section 4 compares and contrasts two implementations of this application based on the CORBA 2.3 and CORBA 2.4 specifications; Section 5 evaluates the results of benchmarks that measure efficiency, predictability, and scalability of key CORBA real-time and messaging features in TAO; Section 6 compares our research on TAO with related work; and Section 7 presents concluding remarks.

2 Synopsis of CORBA 2.3 and 2.4 Features

This section describes the CORBA reference model and highlights the difference between CORBA 2.3 (and earlier CORBA specifications) and CORBA 2.4, focusing on features pertaining to quality of service (QoS).

2.1 The CORBA 2.3 Reference Model

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [4]. Figure 1 illustrates the key components in the CORBA reference model upto and including CORBA 2.3 [23] that collaborate to provide this degree of portability, interoperability, and transparency.[1] Each component in the CORBA reference model is outlined below:

Client: An application plays the client *role* if it obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a

[1]This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [2].

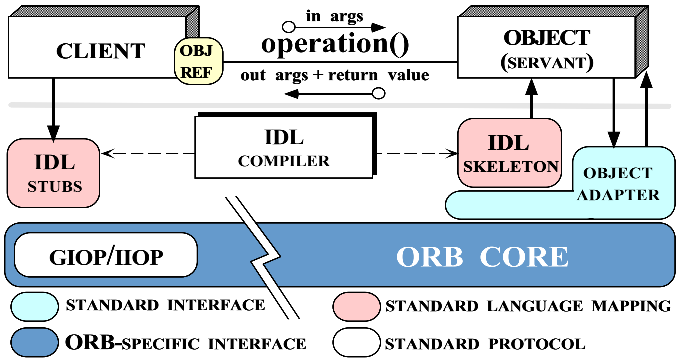

Figure 1: Key Components in the CORBA 2.3 Reference Model

local object, *i.e.*, `object`$\rightarrow$`operation(args)`. Figure 1 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object. Applications can play both the client and server roles.

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *object ID* associates an object with its implementation, called a servant. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `struct`s. A client never interacts with servants directly, but always through objects identified by object references.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is usually implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA interoperability compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [10] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern [10] and demarshal the message-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [24].

Object Adapter: An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

2.2 QoS-related Enhancements to CORBA 2.4

CORBA specifications upto and including CORBA 2.3 [23] lacked features that allow applications to allocate, schedule, and control key CPU, memory, and networking resources necessary to ensure end-to-end quality of service. The CORBA 2.4 standard [2] includes the Messaging [19] and Real-time CORBA specifications [18] that support many of these features. The Messaging specification defines asynchronous operation models [21] and a QoS framework that allows applications to control many end-to-end ORB policies. The Real-time CORBA specification defines interfaces and policies for managing ORB processing, communication, and memory resources. Figure 2 illustrates how these various CORBA 2.4 features interact.

As shown in Figure 2 an ORB endsystem [5] consists of network interfaces, operating system I/O subsystems and communication protocols, and CORBA-compliant middleware components and services. The CORBA 2.4 specification identifies capabilities that must be *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, peer-to-peer) integrated and managed by ORB endsystems to ensure end-to-end predictable behavior for *activities*[2] that flow between CORBA clients and servers.

[2]An activity represents the end-to-end flow of information between a client and its server that includes the request when it is in memory, within the transport, as well as one or more threads.

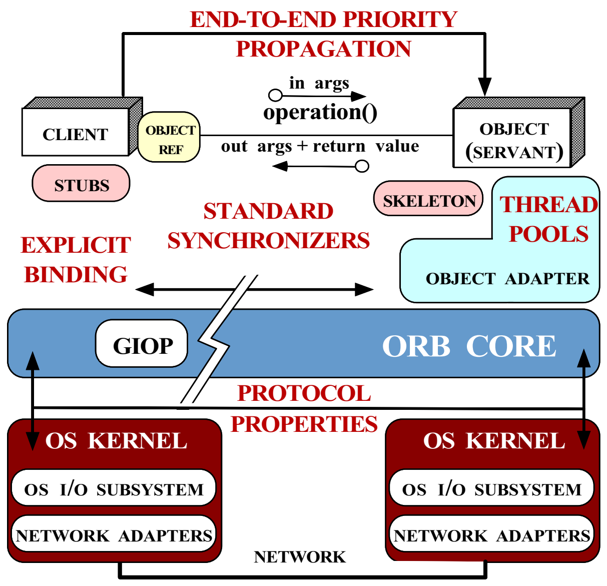

Figure 2: CORBA 2.4 QoS Support for Real-Time Applications

Below, we outline these capabilities, starting from the lowest level abstraction and building up to higher-level services and applications.

1. Communication infrastructure resource management: A CORBA 2.4 endsystem must leverage policies and mechanisms in the underlying communication infrastructure that support resource guarantees. This support can range from (1) managing the choice of the connection used for a particular invocation to (2) exploiting advanced QoS features, such as controlling the ATM virtual circuit cell pacing rate [25].

2. OS scheduling mechanisms: ORBs exploit OS mechanisms to schedule application-level activities end-to-end. The real-time CORBA features in CORBA 2.4 target fixed-priority real-time systems [26]. Thus, these mechanisms correspond to managing OS thread scheduling priorities. The Real-time CORBA specification in CORBA 2.4 focuses on operating systems that allow applications to specify scheduling priorities and policies. For example, the real-time extensions in IEEE POSIX 1003.1c [27] define a static priority FIFO scheduling policy that meets this requirement.

3. Real-Time ORB endsystem: ORBs are responsible for communicating requests between clients and servers transparently. A real-time ORB endsystem must provide standard interfaces that allow applications to specify their resource requirements to the ORB. The QoS policy framework defined by the OMG Messaging specification [19] allows applications to configure ORB endsystem resources, such as thread priorities, buffers for message queueing, transport-level connections, and network signaling, in order to control ORB behavior.

4. Real-time services and applications: Having a real-time ORB manage endsystem and communication resources only provides a partial solution. Real-time CORBA ORBs must also preserve efficient, scalable, and predictable behavior end-to-end for higher-level services and application components. For example, a global scheduling service [5, 28] can be used to manage and schedule distributed resources. Such a scheduling service can interact with an ORB to provide mechanisms that support the specification and enforcement of end-to-end operation timing behavior. Application developers can then structure their programs to exploit the features provided by the real-time ORB and its associated higher-level services.

To manage the ORB endsystem capabilities outlined above, CORBA 2.4 defines standard interfaces and QoS policies that allow applications to configure and control the following resources:

- *Processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service;
- *Communication resources* via protocol properties and explicit bindings; and
- *Memory resources* via buffering requests in queues and bounding the size of thread pools.

Applications can specify these CORBA 2.4 QoS policies along with other policies when they call standard ORB operations, such as `validate_connection` or `create_POA`. For instance, when an object reference is created using a QoS-enabled portable object adapter (POA), the POA ensures that any server-side policies that affect client-side requests are embedded within a *tagged component* in the object reference. Tagged components are name/value pairs that can be used to export attributes, such as security or QoS values, from a server to its clients within object references [2]. Clients who invoke operations on such object references implicitly use the tagged components to honor the policies required by the target object.

3 Meeting the Demands of Real-time, Embedded Applications

3.1 Design Forces

An important class of distributed and embedded real-time applications require stringent real-time end-to-end QoS support. For example, hot rolling mills, sophisticated medical modalities, avionics mission computers, and complex command & control systems share the following characteristics [28]:

Scheduling assurance prior to run-time: Failure to meet deadlines can result in loss of life or significant loss of property. Therefore, the system must be analyzed off-line to ensure that no timing failures will occur at run-time.

Stringent resource limitations: The hardware is typically less powerful than that available for desktop and server computers due to power restrictions, the need to work in hostile environments, and/or the complexity and cost of validation processes.

Distributed processing: To overcome the CPU constraints outlined above, and to improve availability and fault tolerance, real-time embedded systems may be composed of multiple CPUs, each with its own local memory. These CPUs communicate via buses, such as compact PCI or VME, and/or networking hardware, such as ATM or Fibrechannel.

3.2 Avionics Application Scenario

3.2.1 Key Requirements

To evaluate the pros and cons of the CORBA 2.4 specification, we describe how they are being applied in practice to develop an avionics mission computing [6] system. The characteristics of this system are representative of the distributed and embedded real-time application requirements outlined in Section 3.1.

In this context, an avionics mission computer manages sensors and operator displays, navigates the aircraft's course, and controls weapon release, within a distributed airframe environment. DOC middleware for avionics mission computing applications must support deterministic mission computing real-time tasks that possess stringent deadlines. In addition, avionics middleware must support tasks, such as built-in-test and low-priority display updates, that can tolerate minor fluctuations in scheduling and reliability guarantees, but nonetheless require QoS support [28].

While *in theory* it is possible to develop solutions satisfying these requirements from scratch, contemporary economic and organizational constraints are making it implausible to do so *in practice*. It is therefore necessary to integrate (rather than develop from scratch) next-generation avionics systems, using pre-existing COTS hardware and software components whenever possible. COTS middleware is thus playing an increasingly strategic role in the development of many types of software-intensive, mission-critical, and real-time systems.

3.2.2 System Hardware Configuration

Figure 3 depicts a distributed and embedded real-time avionics mission computing configuration [6]. In this scenario, there are three CPU boards interconnected via a VME bus. CPU board 1 has a 1553 interface to communicate with remote terminals such as aircraft sensors for determining global position, inertial navigation, and forward-looking infrared radar [29].

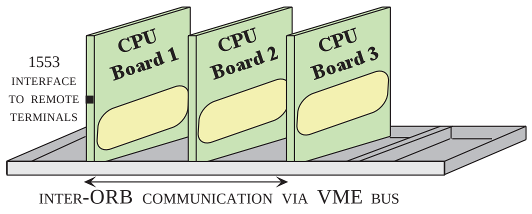
CPU Board 1
CPU Board 2
CPU Board 3
1553 INTERFACE TO REMOTE TERMINALS
INTER-ORB COMMUNICATION VIA VME BUS

Figure 3: A Real-time Embedded Avionics Platform

Avionic mission computers typically collect aircraft sensor data at periodic rates, such as 30 Hz (*i.e.*, 30 times a second), 15 Hz, and 5 Hz. When data is available from a sensor, the mission computer is notified of the event via a hardware interrupt. These sensor-initiated events can trigger subsequent application-generated events that designate specific control or monitoring operations, such as weapons targeting and heads-up display (HUD) processing. Generally, the processing triggered by the application-generated events has stringent completion deadlines that must be enforced throughout the distributed mission computing system.

3.2.3 Key Design Challenges

Developing a solution that provides end-to-end QoS guarantees requires careful consideration of operational requirements. To enforce end-to-end application QoS guarantees, for instance, DOC middleware for avionics mission computing must support the application architecture described above by (1) preserving priorities end-to-end, (2) supporting reliable asynchronous communication, and (3) ensuring sufficient operation throughput [30], as outlined below.[3]

Preserving priorities end-to-end: A key DOC middleware design challenge is to preserve the priority of application requests end-to-end so that consumers can process all the published events by their deadlines. In particular, the client thread that sends a request, the ORB Core thread that receives incoming requests, and the ultimate application server thread that processes the requests must run at the same priority. To ensure this constraint is met, off-line rate monotonic scheduling (RMS) analysis is performed to determine a feasible schedule. Application developers then apply the resulting schedule to statically map periodic events, activities, and ORB threads to fixed real-time priorities, based on the rate at which operations execute.

[3]Other key design challenges include constant-time request demultiplexing [15], integrating I/O subsystems [31] and ORB pluggable protocol frameworks [32], static [5] and dynamic [28] operation and event scheduling. We do not examine these topics in detail in this paper because our other papers describe solutions to these challenges.

Achieving Reliable asynchronous communication: To simplify end-to-end system scheduling and design, it is common for real-time clients and servers to communicate asynchronously. This strategy decouples clients from servers to increase concurrency and prevent unnecessary client blocking. Depending on the underlying transport mechanism, however, it may be necessary to use some form of end-to-end acknowledgment to ensure that requests are delivered reliably across the communication medium.

For example, VME buses are designed to provide reliable transport between boards on the same bus. Inter-board requests can be lost, however, due to buffer mismanagement, overflow, or transient VME bus anomalies that occur under heavy load. In systems where such failures must not go undetected, the use of a higher-level reliability protocol above the transport layer is required.

Ensuring adequate operation throughput: Aircraft sensor devices, such as navigation devices and radar sensors, generate data at regular periodic intervals [6]. Real-time applications may be limited to relatively low operation throughput per-period, however, due to per-operation overhead. This overhead can arise from OS I/O subsystem and network interface processing, as well as from the propagation delay across the communication media.

Fortunately, due to the periodic nature of certain real-time applications, it is possible to delay some requests, while still meeting overall timing constraints. For example, display updates can be deferred until shortly before their deadlines. Such requests can be buffered and sent in a single data transfer, thereby minimizing transport overhead and increasing operation throughput.

3.2.4 Evaluating Avionics Application Software Architectures

Overview: Figure 4 shows a conventional non-CORBA architecture for distributing periodic I/O events throughout an avionics application. This example has the following participants:

• **Aircraft sensors:** Aircraft-specific devices generate sensor data at regular intervals, *e.g.*, 30 Hz, 15 Hz, 5 Hz, *etc*. The arrival of sensor data generates interrupts that notify the mission computing applications to receive the incoming data.

• **Sensor proxies:** Mission computing systems must process data to and from many types of aircraft sensors, including global position system (GPS), inertial navigation set (INS), and forward looking infrared radar. To decouple the details of sensor communication from the applications, sensor proxy objects are created for each sensor on the aircraft. When I/O interrupts occur, data from a sensor is given to an appropriate sensor proxy. Each sensor proxy object demarshals the incoming data and notifies I/O facade objects that depend on the sensor's data. Since modern aircraft can be equipped with hundreds of sensors, a large number of sensor proxy objects may exist in the system.

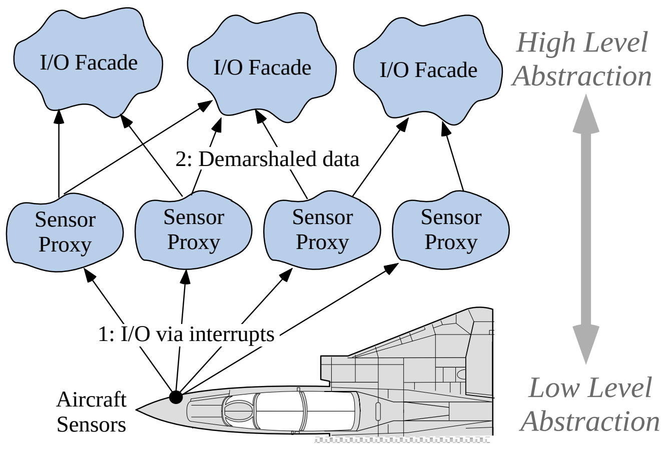

Figure 4: Example Avionics Mission Control Application

• **I/O facades:** I/O facades represent objects that depend on data from one or more sensor proxies. I/O facade objects use data from sensor proxies to provide higher-level views to other application objects. For instance, the aircraft position computed by an I/O facade is used by the navigation and weapons release subsystems.

The *push*-driven model described above is commonly used in many real-time environments [33], such as industrial process control systems and military command/control systems. One positive consequence of this push-driven model is its efficient and predictable execution of operations. For instance, I/O facades only execute when their event dependencies are satisfied, *i.e.*, when they are called by sensor proxies.

In contrast, using a *pull*-driven model to design mission applications would require I/O facades that actively acquire data from sensor proxies. If data were not available to be pulled, the calling I/O facade must block awaiting a result. Thus, for I/O facades to pull, the system must allocate additional threads to allow the application to progress while I/O facade tasks block. Adding threads to the system has many negative consequences, however, such as increased context switching overhead, synchronization complexity, and complex real-time thread scheduling policies [22]. Conversely, by using the push model, blocking is largely alleviated, which reduces the need for additional threads. Therefore, this paper focuses on the push model.

Drawbacks with conventional avionics architectures: A disadvantage to the architecture shown in Figure 4 is the strong coupling between suppliers (sensor proxies) and consumers (I/O facades). For instance, to call back to I/O facades, each sensor proxy must know which I/O facades depend on its data. As a result, changes to the I/O facade layer, such as adding/removing consumers, require sensor proxy modifications. Likewise, consumers that register for callbacks are tightly coupled with suppliers. If the availability of new hardware, such as forward looking infrared radar, requires a new sensor proxy, the I/O facades must be altered to take advantage of the new technology.

Alleviating drawbacks with the CORBA Event Service: Figure 5 shows how the CORBA Event Service [34] can help

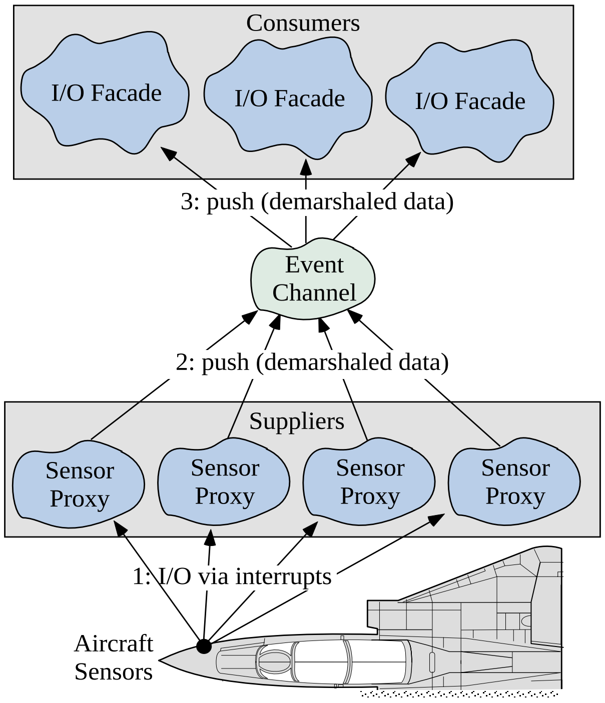

Figure 5: Example Avionics Application with CORBA Event Channel

alleviate the disadvantages of the tightly coupled consumers and suppliers shown in Figure 4. The CORBA Event Service defines *supplier*, *consumer*, and *event channel* participants so that distributed applications can exchange requests asynchronously via an *event-based* execution model [33]. Suppliers generate events, consumers process events sent by suppliers, and event channels propagate events to consumers on behalf of suppliers.

The architecture of our example avionics mission computing application [6] centers on the Publish/Subscribe [11] pattern shown in Figure 5. As shown in this figure, sensors generate an interrupt to indicate data availability. Sensor proxies then push the event to an event channel, which dispatches events to application-level consumers on behalf of the sensors generating the events. When consumers subscribe with an event channel, they indicate what types of events they are interested in receiving by supplying the filtering criteria. The benefit of using an event channel is that sensor proxies are unaffected when I/O facades are added or removed.

Before running production mission computing applications, the system is analyzed and events are assigned priorities by a real-time (RT) scheduling service [5]. Since the same consumer can receive events at different priorities, the DOC middleware must support this use-case, thereby allowing applications to use the same object reference to access a service at different priority levels.

Another benefit of a CORBA event channel-based architecture is that an I/O facade need not know which sensor proxies supply its data. Since the channel mediates on behalf of the sensor proxies, I/O facades can register for certain types of events (*e.g.*, GPS and/or INS data arrival) without knowing which sensor proxies actually supply these types of events. Once again, the use of an event channel makes it possible to add or remove sensor proxies without changing I/O facades.

4 Evaluating CORBA for Real-time Applications

During the past several years, we developed two variants of the real-time avionics mission computing application [6] described in Section 3.2. Each variant used a different version of TAO based on different versions of the CORBA specification, The initial variant was based on CORBA 2.3 [23] and is outlined in Section 4.1. The later variant was based on the Real-time [18] and Messaging [19] specifications defined in CORBA 2.4 and is described in Section 4.2.

4.1 Synopsis of the Original CORBA 2.3-based Solution

4.1.1 Overview

Our original solution for the avionics mission computing application described in Section 3.2 was developed several years ago using the CORBA 2.3-based version of TAO ORB and TAO's Real-time (RT) Event Service [6] and RT Scheduling Service [5]. This solution was developed before the OMG adopted the CORBA 2.4 Real-time and Messaging specifications. Therefore, it supported real-time mission computing applications via several non-standard CORBA extensions, such as pre-establishing connections [22] to particular connection endpoints and allowing multiple ORBs to share a single object adapter to simplify object activation across rate groups, as shown in Figure 6.

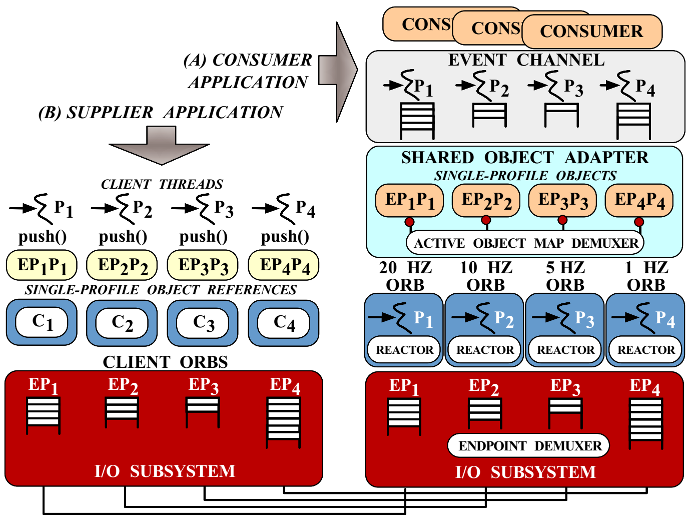

Figure 6: TAO's Original CORBA 2.3 Solution

In the original CORBA 2.3 solution, multiple supplier threads on one CPU board ran at different priorities, *e.g.*, $P_1 \ldots P_4$. One-way operations were used to push events to consumer threads that ran at corresponding priorities on other CPU boards. Each ORB on the server created a `Reactor` [14]. This component was responsible for accepting connections at a particular connection endpoint, demultiplexing connections at the associated ORB's priority level, and dispatching incoming client requests to ORB Core connection handlers.

During off-line system scheduling, priorities were calculated for each of the rate groups using TAO's static RT Scheduling Service [5]. Then, during on-line system initialization, a thread for each calculated priority was spawned and multiple ORBs – one per thread – were created on each CPU board and associated with a particular connection endpoint, *e.g.*, EP_1 ... EP_4. This configuration enabled each ORB to process events associated with its thread's rate group. Finally, all the ORBs on each CPU board pre-established connections, *e.g.*, C_1 ... C_4, to all other ORBs listening on connection endpoints on other CPU boards.

The mission computing application software itself was responsible for mapping each request onto the appropriate ORB/connection in accordance with its priority. This design ensured that individual invocations and event channel processing occurred at the correct real-time priorities. For example, if a consumer processed a 20 Hz event, it used its 20 Hz ORB and the object references generated by server ORBs that correspond to the thread priority (P_1) of the 20 Hz rate group. Moreover, by statically configuring all inter-ORB connections and mapping them to specific thread priorities, this design avoided head-of-line-blocking in the socket connection queues.

4.1.2 Evaluation of the CORBA 2.3-based Solution

The CORBA 2.3-based solution described above met its original objectives, *i.e.*, it minimized key sources of unbounded priority inversion and was flown successfully in an operational test-flight [35]. However, this solution had the following drawbacks:

Overly complex and non-standard programming model: Maintaining multiple ORBs to preserve end-to-end priority turned out to be hard for mission computing application developers. In the server, an ORB in each thread had to be initialized. Moreover, servants had to be activated multiple times, *i.e.*, once under each ORB, to obtain a different object reference for each rate group. In the client, the application kept multiple object references for each object, one for each rate group. Moreover, the application was responsible for managing the association between these object references and priorities, *e.g.*, using an *ad hoc* location service.

Excessive memory consumption: Each ORB maintained a separate copy of its resources, such as its active object map and its connection cache. Although this design minimized contention and priority inversion, it required excessive duplication of resources. For example, there were as many copies of active object maps as there were threads because each servant was activated on each ORB.

Inefficient initialization: In addition to consuming excessive memory resources, the use of multiple ORBs also increased the time required to initialize the avionics system. This overhead is particularly problematic because the avionics mission computers reinitialize rapidly to recover from transient power cycles, *e.g.*, due to sudden aircraft deceleration during carrier landings.

Tight coupling between threads and ORBs: Each thread contained its own instance of an ORB, as shown in Figure 6. Although this design minimized synchronization within the ORB, it hampered the usability of the system by application developers. For instance, object references created in one ORB could be used only in the thread of that ORB. Therefore, application developers could not pass object references between threads on the same CPU board because client/servant collocation did not work transparently [15].

Ambiguous one-way invocation semantics: The semantics of one-way method invocations in CORBA 2.3 were ambiguous and could be implemented differently by different ORBs [4]. Therefore, applications could not control the reliability of requests, nor could they control overall system throughput.

4.2 Synopsis of the CORBA 2.4-based Solution

4.2.1 Overview

To address the limitations of the solution described in Section 4.1, the avionics mission computing application was reimplemented using a version of TAO that is based on the CORBA 2.4 Real-time [18] and Messaging [19] specifications. Below, we describe and evaluate how well suited CORBA 2.4 features, such as *priority preservation* and *reliable one-ways*, are to meeting key challenges of this class of applications. For each set of challenges we outline the changes made to TAO during this effort and motivate several enhancements we developed to provide more precise control over the CORBA 2.4 real-time and messaging features.

4.2.2 Preserving Priorities End-to-End

Context: Systems with stringent QoS requirements, such as our avionics mission computing application, often must execute a request at the same priority, end-to-end, as described in Section 3. In the following paragraphs we outline the Real-time CORBA mechanisms in CORBA 2.4 intended to preserve request priorities end-to-end.

- **Priority mapping:** The specification defines a universal, platform-independent priority representation called the *CORBA Priority*. This feature allows applications to make prioritized CORBA invocations in a consistent fashion between nodes running on operating systems with different priority schemes. *Priority mapping functions* are used to map priority values specified in terms of *CORBA priority* into *native OS priority*.

- **CORBA priority models:** The Real-time CORBA specification defines a *PriorityModel* policy that determines the priority at which server handles requests from clients. The policy can have one of the two values: SERVER_DECLARED or CLIENT_PROPAGATED. In the SERVER_DECLARED model shown in Figure 7 (A), the server handles requests at the priority declared on the server side at object creation time. This priority is communicated to the client in an object reference.

The Real-time CORBA specification also defines the CLIENT_PROPAGATED model shown in Figure 7 (B). In this model, the client encapsulates its priority in the service context list of the operation invocation and the server then honors the priority of the invocation. When a server ORB parses the request, it extracts the priority from the service context and sets the priority of the processing thread to match the requested priority.

- **Thread pools:** A Real-time CORBA server can associate each POA with a pool of pre-allocated threads running at appropriate priorities. A pool can optionally be pre-configured for a maximum buffer size or number of requests, as shown in

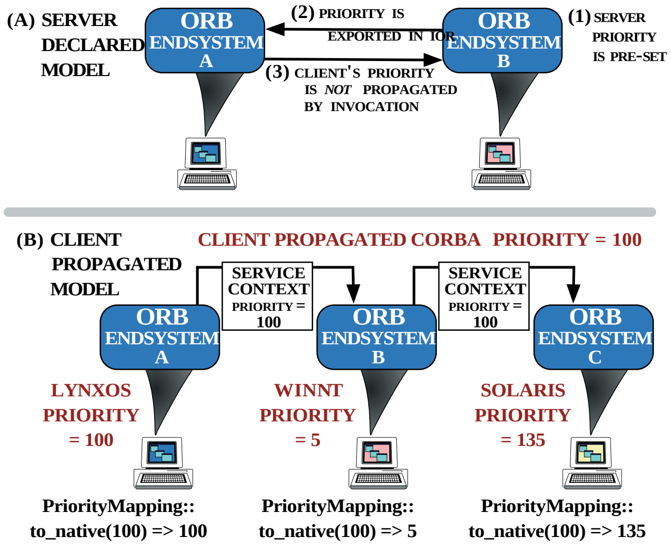
(A) SERVER DECLARED MODEL
ORB ENDSYSTEM A
(2) PRIORITY IS EXPORTED IN IOR
(3) CLIENT'S PRIORITY IS NOT PROPAGATED BY INVOCATION
ORB ENDSYSTEM B
(1) SERVER PRIORITY IS PRE-SET
(B) CLIENT PROPAGATED MODEL
CLIENT PROPAGATED CORBA PRIORITY = 100
SERVICE CONTEXT PRIORITY = 100
SERVICE CONTEXT PRIORITY = 100
ORB ENDSYSTEM A
ORB ENDSYSTEM B
ORB ENDSYSTEM C
LYNXOS PRIORITY = 100
WINNT PRIORITY = 5
SOLARIS PRIORITY = 135
PriorityMapping:: to_native(100) => 100
PriorityMapping:: to_native(100) => 5
PriorityMapping:: to_native(100) => 135

Figure 7: Real-time CORBA Priority Models

Figure 8. If buffering is enabled for the pool, the request will

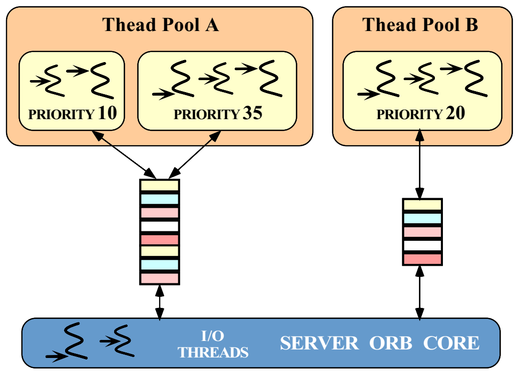

Figure 8: Buffering Requests in Real-time CORBA Thread Pools

be queued until a thread is available to process it. If no queue space is available or request buffering was not specified the ORB should raise a TRANSIENT exception, which indicates a temporary resource shortage. When the client receives this exception it can reissue the request at a later point.

- **Priority banded connections:** This feature allows a client to communicate with the server via multiple transport connections. Each connection is dedicated to carrying invocations of distinct CORBA priority or range of priorities, as shown in Figure 9. A client ORB establishes a priority banded connection by sending a server the `_bind_priority_band` request, which specifies the range of priorities the connection will be used for. This feature allows the server to allocate the necessary resources for the connection and to configure these resources to provide service for the specified priority range.

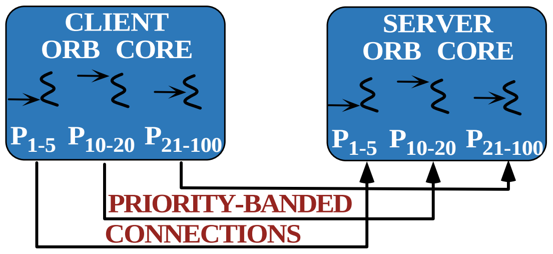

Figure 9: Priority Banded Connections

The selection of the appropriate connection for each invocation is transparent to the application, and is done by the ORB based on the value of the *PriorityModel* policy.

Problems: As outlined above, the mechanisms defined in the Real-time CORBA chapter of the CORBA 2.4 specification provide application developers with greater control over ORB endsystem resources than earlier CORBA 2.3 specifications. For many real-time applications these mechanisms are sufficient to provide the necessary QoS guarantees. For real-time applications with stringent QoS requirements such as those outlined in Section 3, however, this lack of specificity can lead to ineffective or *non-portable* implementations, as discussed below:

- **Priority mapping problems:** Although Real-time CORBA mandates each ORB to provide default priority mapping functions, as well as a mechanism to allow users to override these defaults, it does not state how those mappings functions are accessed and set. Thus, application developers are forced to use proprietary interfaces.

- **CORBA priority model problems:** The Real-time CORBA CLIENT_PROPAGATED model can be inappropriate for applications with hard real-time requirements due to opportunities for priority inversion [36]. In particular, it is possible that the initial priority of the thread reading the request is too high or too low, relative to the priority of the thread that processes the servant in an upcall.

Likewise, the SERVER_DECLARED priority model is not appropriate for applications that invoke the *same* operation on the *same* object, but at *different* priorities from *different* client threads. For example, if our avionics mission computing application were to use the SERVER_DECLARED priority model, it would have to activate the same servant multiple times, using a different priority for each activation. The client application would then choose the object reference based on the client thread's priority, and invoke the operation on the right object. However, this solution is unnecessarily complicated for the following reasons:

- It would interact poorly with CORBA location services, such as Naming or Trading, because each object must be registered multiple times.
- An application-specific client convention would be required to (1) fetch all the object references for the same object and (2) map priorities to the corresponding object references.

In our CORBA 2.3-based implementation, we faced similar challenges in managing multiple object references corresponding to multiple server ORBs. Our experiences indicated that such an approach yielded complex code that was hard to maintain, thereby negating several advantages of DOC middleware.

- **Thread pool problems:** The Real-time CORBA specification does not provide any policies to ensure that threads in a pool receive requests directly from connections. Thus, a compliant implementation may choose to separate threads that perform all the I/O, parse the request to identify the target POA and priority, and hand off the request to the appropriate thread in the POA thread pool, as shown in Figure 10. Such an implementation can increase average and worst-case

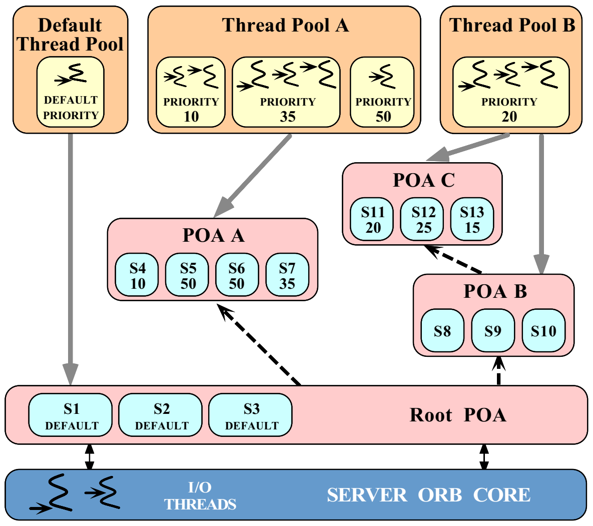

Figure 10: An Inappropriate CORBA Thread Pool Architecture for Hard Real-time Applications

latency and create opportunities for unbounded priority inversions [15], however. For instance, even under a light load, the server ORB incurs a dynamic memory allocation, multiple synchronization operations, and a context switch to pass a request between a network I/O thread and a POA thread.

- **Priority banded connection problems:** There is no standard API in Real-time CORBA that allows server applications to control how thread pools are associated with priority banded connections. For instance, a server application can not control whether its ORB assigns each connection a separate thread, or whether a pool of threads can be pre-allocated to service multiple connections that have the same priority range. Unfortunately, this lack of detail in the specification makes it hard to write real-time applications that behave predictably across different ORB platforms.

The Real-time CORBA specification also lacks a standard API that would allow a server application to control how its ORB associates a thread at a pre-specified priority to read requests from a priority banded connection. Thus, the actual ORB thread that performs I/O operations could be different from the thread processing the request, and could execute at the wrong priority, thereby incurring priority inversion. This lack of specificity in the Real-time CORBA priority banded connections mechanism can lead to implementations that suffer from problems similar to those with POA thread pools shown in Figure 10.

Solution → Prioritized connection endpoints: To alleviate the problems listed above, we defined mechanisms in TAO to explicitly map thread pools and thread priorities to connection endpoints. These mechanisms extend the Real-time CORBA specification to give TAO applications greater control over the mapping of connections to thread priorities within the ORB Core. Figure 11 provides an example of the avionics mission computing application shown in Figure 6 revised to use TAO's prioritized connection endpoints mechanism.

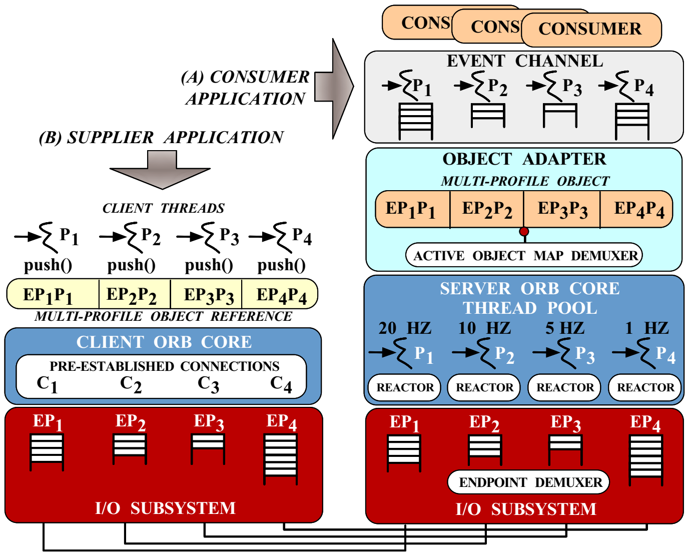

Figure 11: TAO's New CORBA 2.4-based Solution

As shown in this figure, the server application is an event channel consumer with four connection endpoints, EP_1 ... EP_4. Each endpoint is assigned a CORBA priority, *e.g.*, EP_1 has priority $P1$, and is serviced by a thread of the corresponding native OS priority. Object references for servants activated in this server contain four profiles, one for each endpoint, as shown in the object adapter portion of the server in Figure 11 (A).

The client application in Figure 11 (B) is an event channel supplier. It has four threads with priorities $P1 \ldots P4$. When a supplier thread makes a call on the consumer object reference exported by the server, the ORB finds a pre-established connection for that endpoint and uses it to send the request. The priority is preserved end-to-end because on the server-side the connection is serviced by a thread at the same priority as the thread making the request on the client-side.

Implementing prioritized connection endpoints in TAO: Below, we describe key mechanisms provided by TAO to implement prioritized connection endpoints in client and server ORBs:

• **Server ORB support for binding thread priorities to listen-mode connection endpoints:** TAO allows servers to have multiple listen-mode [37] connection endpoints, each associated with a CORBA priority. Each connection endpoint is also statically associated with a pool of threads running at native OS priority corresponding to the CORBA priority of the endpoint. Pool threads are responsible for accepting and servicing connections on the associated endpoint.

When an object is activated in a server with multiple endpoints, the generated object reference contains multiple profiles, one for each endpoint. Each profile stores the CORBA priority of its endpoint in a tagged component. This design allows a client to receive service at its desired priority by simply selecting and using the profile containing that priority.

TAO's prioritized connection endpoints extension is particularly attractive for applications, such as avionics mission computing, that invoke operations on the same object at different priority levels. In particular, these applications can specify a set of prioritized connection endpoints on the command-line, effectively defining the set of priorities supported on the server *a priori*, thereby allowing the ORB to schedule and allocate resources more effectively end-to-end. Moreover, this programming model is much simpler than creating multiple objects and object references and trying to assign them different thread priorities.

• **Client ORB support for connections with priorities:** When a client makes an invocation, the client-side ORB must select one of the profiles from an object reference before sending the request to the server. The profile is selected based on the priority at which the client wants the request serviced. To allow clients to specify this desired priority, TAO defines a *ClientPriority* policy. Clients can set the *ClientPriority* policy to one of the following values:

- USE_NO_PRIORITY – *i.e.*, priority information is not used when a client ORB selects a profile from an object reference.
- USE_THREAD_PRIORITY – *i.e.*, the priority of the client thread sending a request is used to select the profile. This option is used when the priority of request must be preserved end-to-end. For example, we use this option in

Figure 11 (B), where the *ClientPriority* policy is checked before the client ORB selects a profile from the object reference. In that case, when a client thread with priority $P1$ invokes an operation on a consumer object, the ORB selects the profile corresponding to that priority, *i.e.*, the one that contains connection endpoint EP_1.

- USE_PRIORITY_RANGE – In this case, a range of priorities to be used for profile selection is specified by the application inside the policy. This option allows applications to request services at a priority that is *different* than that of the client thread invoking an operation. For example, a high-priority client thread can generate a low-priority event, such as a display update. This high-priority thread can post the message in a remote server at a low-priority, to minimize the effect on more critical processing. However, it need not change its own priority to perform this task, which avoids local priority inversions.

TAO's prioritized connection endpoints and *ClientPriority* policy extend the standard Real-time CORBA priority models and its priority banded connections mechanism to achieve an effective balance between the SERVER_DECLARED and CLIENT_PROPAGATED models. In particular, TAO provides the same degree of control to the server as the SERVER_DECLARED model by restricting clients to use well-known priorities. However, it also allows clients to select a priority published by the server that best meets their requirements. TAO's design avoids priority inversions and ensures ORB endsystem resources are strictly controlled, while still retaining a simple programming model.

Section 5.1 illustrates the performance of TAO's prioritized connection endpoint architecture.

4.2.3 Achieving Reliable Asynchronous Communication

Context: Embedded real-time CORBA applications often use one-way operations to simulate message-passing via standard CORBA features. For example, avionic mission computing applications [6] process periodic event messages, such as sensor updates and heartbeat messages from redundant systems. Typically, clients send these messages to servers via CORBA one-way operations, which require no response.

Problems: The semantics of conventional CORBA one-way operations are often unacceptable because the CORBA 2.3 specification does not require an ORB to guarantee that one-way operations will be delivered [4].

Solution $\rightarrow$ CORBA 2.4 reliable asynchronous features: To alleviate the problem outlined above, the CORBA Messaging specification defines a policy called *SyncScope* that allows clients more control over the degree of reliability for one-way operation invocations. Figure 12 illustrates the following four levels of reliability for one-way operations:

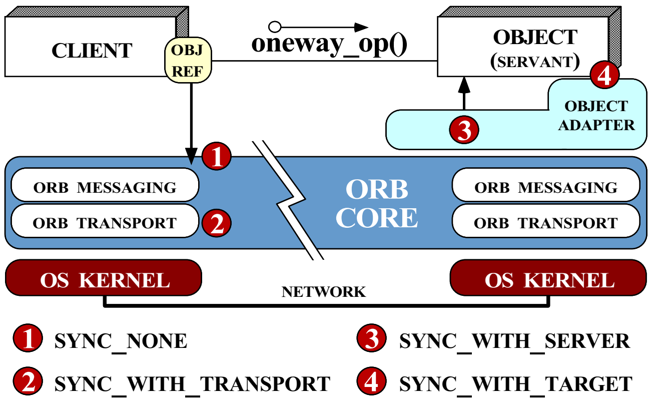

Figure 12: Reliable One-way Synchronization Scopes

- **SYNC_NONE:** With this policy value, the client ORB returns control to the client application before passing the request to the transport layer. This value minimizes the amount of time a client spends blocking on the one-way operation, but provides the lowest level of delivery guarantee. The SYNC_NONE policy is useful for applications that require minimal client operation latency, while tolerating reduced reliability guarantees.

- **SYNC_WITH_TRANSPORT:** With this policy value, the ORB returns control to the client only after the request is passed successfully to the transport layer, *e.g.*, the client's TCP protocol stack. A client can incur unbounded latencies if a connection endpoint flow controls due to a limited buffer space. When used with a connection-oriented transport, such as TCP, SYNC_WITH_TRANSPORT can provide more assurance than SYNC_NONE. This policy is appropriate for clients that require a compromise between low latency and reliable delivery.

- **SYNC_WITH_SERVER:** With this policy value, the client invokes a one-way operation and then blocks until the server ORB sends an acknowledgment. The server ORB sends the acknowledgment after invoking any servant managers, but before dispatching the request to the servant. The SYNC_WITH_SERVER policy value provides clients with assurance that the remote servant has been located. This feature is particularly useful for real-time applications that require some degree of reliability, *e.g.*, because they run over backplanes that lose packets occasionally, but need not wait for the entire servant upcall to complete.

- **SYNC_WITH_TARGET:** This policy value is equivalent to a synchronous two-way CORBA operation, *i.e.*, the client will block until the server ORB sends a reply after the target object has processed the operation. If no exceptions are raised, the client can assume that the target servant processed its request. This synchronization level is appropriate for clients that need assurance that the upcall was performed and can tolerate the additional latency.

Implementing reliable one-ways in TAO: The *SyncScope* policy controls the reliability of one-way requests. It can be set at the object-level, thread-level, or ORB-level. As with any CORBA policy, the more specific levels can override the more general levels. To implement reliable one-way requests, TAO's IDL compiler [38] generates client stub code that checks the *SyncScope* policy value and sets the appropriate bits in the `response_flags` field in the GIOP request header.

If the *SyncScope* policy is SYNC_NONE, the request is buffered, as described in Section 4.2.4. If the policy value is SYNC_WITH_SERVER or SYNC_WITH_TARGET, the client ORB must wait for a reply from the server and check for a LOCATION_FORWARD response or a CORBA system exception.

On the server, the ORB's behavior is based solely on the value of the `response_flags` field of the request header. If the flags are set to a *SyncScope* policy value of SYNC_WITH_TARGET, the request is treated as a two-way request, whether it originated as a one-way or as a two-way. If the flags are set to a value of SYNC_WITH_SERVER, however, a response will be initiated by the Object Adapter immediately after it locates the servant, but before dispatching the upcall.

Section 5.3 presents benchmarks illustrating the performance of TAO's reliable one-way implementation.

4.2.4 Ensuring Adequate Operation Throughput

Context: Distributed real-time applications often have stringent timing requirements, where critical operations must begin and/or complete within specified time intervals. For example, aircraft sensor devices, such as navigation devices and radar sensors, generate data that must be processed at regular periodic intervals [6]. Such applications often have a fixed time period in which to invoke remote one-way operations. After invoking each operation, the client must perform other processing, *i.e.*, it does not wait synchronously for the server to process the operation and respond.

Problems: The following two problems can arise when applying ORB middleware to distributed real-time applications with periodic processing requirements:

• **Inadequate operation throughput:** The time spent delivering a one-way or asynchronous operation to a server includes the overhead of invoking one or more `write` calls to the client OS. In turn, this incurs protocol stack and network interface processing, as well as the propagation delay across the communication media. This per-operation overhead constitutes a non-trivial amount of the total end-to-end latency, particularly for small requests. As a result of this overhead, real-time applications may be limited to a relatively low number of remote operations per time period.

• **Blocking flow control:** It is important that periodic real-time applications not block indefinitely when ORB endsystem and network resources are unavailable temporarily. However, ORB transport protocols, such as IIOP, often implement reliable data delivery using a sliding window flow control algorithm [37]. Thus, they may block the client from transmitting additional data when the communication channel is congested, or if the server is slow. Although some transport protocols buffer a limited number of bytes or requests, they will typically block client threads after this limit is reached.

One way to solve these problems is to revise the application's IDL interfaces and reimplement the clients so they buffer data at the application-level. This solution works well for certain periodic applications that can sacrifice some latency for increased operation throughput. Buffering at the application-level increases the burden on application developers, however, thereby increasing the implementation, validation, and maintenance effort. Moreover, if two or more application IDL interfaces require buffering, code can be duplicated unnecessarily, which increases application footprint.

Solution • ORB-level request buffering: Often, a more effective solution is to have the ORB buffer one-way and asynchronous invocations transparently.[4] At some later time, the buffered requests can be delivered *en masse* to the server. There are several benefits to ORB-level request buffering:

- By buffering requests, a client ORB amortizes the per-operation processing overhead and increases effective network utilization.
- The ORB can use OS *gather write* operations, such as `writev` [37], to minimize the number of mode switches needed to transmit the buffered requests.
- ORB-level buffering can increase application control over the buffering of CORBA requests. This feature is important when the buffering provided by the transport protocol is inadequate, thereby forcing indefinite blocking of the client due to flow control.

Implementing ORB-level request buffering in TAO: The CORBA 2.4 Messaging specification introduces several mechanisms to give application developers more control over QoS parameters than in CORBA 2.3 specifications. In particular, applications can use CORBA 2.4 features, such as the *SyncScope* policy, to control latency/reliability tradeoffs. For example, applications can use the SYNC_WITH_SERVER policy

[4]Synchronous two-way requests and reliable one-way operations should not be buffered. The ORB must deliver these request immediately to the server because the client waits for the server's response before continuing.

value to achieve reliable transport delivery, without waiting for the entire servant's computation to complete. Likewise, the application can ensure non-blocking behavior by using the SYNC_NONE policy value, which TAO implements by buffering multiple requests before sending them to the server.

Unfortunately, neither the CORBA Messaging or the RT CORBA specifications provide mechanisms to control the size or duration of buffers, nor does it provide explicit interfaces to flush buffers. These semantics are insufficient for applications that require precise control over the ORB utilization of memory and network resources. Therefore, we have extended TAO to allow applications to specify multiple strategies for delivering buffered requests via a new *BufferingConstraint* policy.

Figure 13 illustrates how TAO uses this policy to buffer one-way invocations inside the ORB Core for subsequent delivery. When application-specified buffering limits are reached, the buffers are flushed and the queued requests are delivered to the server.

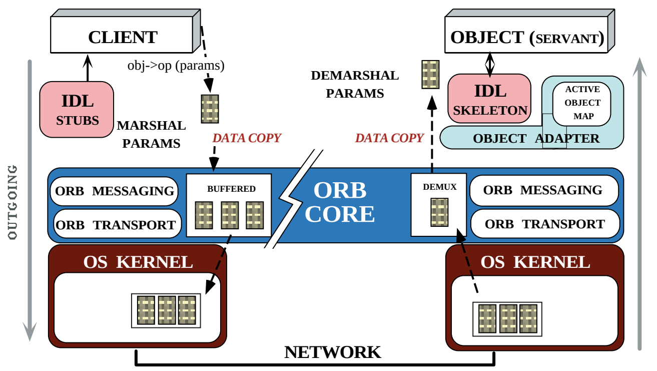

Figure 13: One-way and Asynchronous Request Buffering

A combination of the following conditions can be specified simultaneously using TAO's *BufferingConstraint* policy:

1. Message Count: When the number of buffered messages reaches an application-specified high-water mark, the buffered requests are delivered to the server. This approach allows applications to batch *n* requests together.

2. Message Bytes: When the number of bytes in the buffered messages reaches an application-specified high-water mark, the buffered requests are delivered to the server. This approach allows applications to buffer *n* bytes at the ORB layer.

3. Periodic Timeout: After an application-specified time interval, the ORB delivers any buffered requests to the server. This approach allows applications to pace the delivery of messages to the server even when the requests are produced at irregular intervals.

4. Explicit Flushing: Applications can flush any queued messages explicitly. This approach allows applications to deliver the batched messages to the server in response to some external event.

5. Out-of-Band Requests: Applications can skip buffering for some requests. This approach allows applications to deliver urgent requests to the server immediately, bypassing the buffered requests.

Section 5.2 presents benchmarks illustrating the performance of TAO's buffered requests implementation.

4.2.5 Evaluation of the CORBA 2.4-based Solution

Our CORBA 2.4-based avionics mission computing solution has the following improvements over the original CORBA 2.3-based design:

More standard programming model: The CORBA 2.4 Real-time specification defines a standard model for implementing many features required for avionics mission computing using only a single ORB per CPU. This model supports the mapping of priorities to particular invocations, objects, and threads. As a result, application programming is simplified and the portability of application software increases because the system is based on a standard.

Reduced memory footprint: Our original CORBA 2.3-based solution required multiple ORBs be created within each client and server, once for each rate group. Moreover, servants had multiple object references, one for each global priority. This design resulted in a relatively large application footprint. In contrast, our CORBA 2.4-based solution allows each client and server application to create a single ORB. Thus, only one object reference is created per servant, which further reduces the overall memory footprint.

Efficient initialization: In addition to simplifying the programming model and minimizing the required memory resources, the use of one ORB per-process reduces the time required to initialize the avionics mission computing applications. Reducing this overhead is particularly important when the system must recover from transient power cycles.

Simplified client threading model: Clients are greatly simplified because they manage only one set of object references. Policies and object references contain sufficient information for the ORB to determine the appropriate connection to use on each request.

Improved priority preservation: By supporting multiple connection endpoints within server ORBs, the CORBA 2.4-based implementation has several benefits. For example, the destination service access point, such as the TCP port number, can be mapped to a global CORBA priority thereby ensuring that all CORBA requests within a connection queues have the same priority. This early demultiplexing [31] technique, combined with client and server ORBs' respect of a request's priority, results in *vertically* (*i.e.*, network interface

$\leftrightarrow$ application layer) and *horizontally* (*i.e.*, peer-to-peer) integrated ORB endsystems. The resulting DOC middleware environment preserves invocation priorities end-to-end, *i.e.*, throughout the ORB endsystems and inter-ORB connections.

Improved one-way invocation semantics: Real-time applications must often balance the competing needs of reliable communications, network throughput, and invocation latency. The improved semantics added by the CORBA 2.4 Messaging specification [19] gives clients greater control over these tradeoffs.

In addition to alleviating the drawbacks with our original CORBA 2.3-based solution, the new CORBA 2.4-based version of TAO also provides the following benefits:

Easier integration with CORBA common object services: To use CORBA common object services, such as Naming and Trading, in the CORBA 2.3-based approach, a server must export multiple object references to the same servant, one for each priority. Then, to locate the object reference corresponding to the desired priority, a client must use an *ad hoc* mechanism to retrieve the desired object references at the appropriate priorities. In contrast, in the CORBA 2.4-based approach there is no need for multiple object references. Therefore, no *ad hoc* protocol for mapping priorities to object references is required.

With other services, CORBA 2.3-based approach may require modification to the service itself. For example, the Event Service invokes operations on application-provided objects. To invoke these operations at the appropriate priorities the service must: (1) have access to multiple object references for each application object and (2) select object references corresponding to desired priorities using the application defined protocol. In contrast, in the CORBA 2.4-based approach these tasks are all performed by the ORB transparently to applications and services.[5]

Easier integration with real-time scheduling services: The CORBA 2.4 Real-time specification supports higher-level CORBA scheduling services that allocate resources end-to-end. For example, TAO's static scheduling service [5] can associate application activities with global CORBA priorities. Such scheduling services can use the priority transformations and the policy framework defined in the CORBA Messaging [19] to create sophisticated and adaptive real-time applications.

[5]It is still possible that implementations of these services are unsuitable for real-time application, *e.g.*, due to excessive priority inversion.

5 The Performance of the TAO Real-time CORBA ORB

5.1 Preserving Priorities End-to-End

Overview: The benchmarks in this section compare the performance of TAO's CORBA 2.3-based solution, *i.e.*, the ORB-per-thread approach described in Section 4.1, with TAO's CORBA 2.4-based solution, *i.e.*, using the prioritized connection mechanism described in Section 4.2.2. In particular, to determine how well each approach preserves priorities end-to-end, we compare the latency and jitter of a high-priority client thread as it competes with a variable number of low-priority client threads.

Hardware/OS Benchmarking Platforms: All benchmarks in this section were performed between two 266 MHz PowerPC boards with 32 MBytes of RAM, running the LynxOS 3.0.0 operating system and connected by a 100 Mbps Ethernet. The tests were run with real-time, preemptive, FIFO thread scheduling, which provides strict priority-based scheduling to application threads.

Measurement techniques: Below we describe the client-side and the server-side parts of the benchmark.

- **Client-side:** On the client, a single high-priority thread and a variable number of low-priority threads run concurrently. Both CORBA 2.3-based solution and CORBA 2.4-based solution were benchmarked with 1, 3, 6, 9, 12, and 15 low-priority client threads. Each low-priority thread has a different priority value. The range of LynxOS native priorities used by these threads is 64 to 79. The high-priority thread runs at priority 128.

When the test program creates the client threads, these threads block on a barrier lock so that no client thread starts until the others are created and are ready to run. When all client threads are ready to send requests, the main thread unblocks them. Each client thread issues 20,000 requests to the server at the fastest possible rate.

- **Server-side:** On the server, a servant is created and configured to service client requests at the same priorities as those of its client peers. In the original CORBA 2.3-based approach, this is achieved by creating an ORB-per-thread for each test priority. In the CORBA 2.4-based approach, this is achieved by mapping connection endpoints to threads possessing the appropriate priorities.

Regardless of the approach, each request is serviced at the same priority as that of the invoking client thread. This mapping is achieved using different mechanisms in the two configurations. In CORBA 2.3 configuration, each client thread uses a different object reference, *i.e.*, the one published by the server ORB running in the thread with corresponding priority.

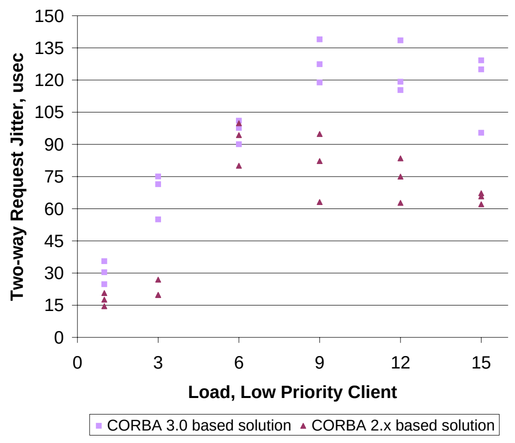

Figure 14: Jitter for High-Priority Client Thread

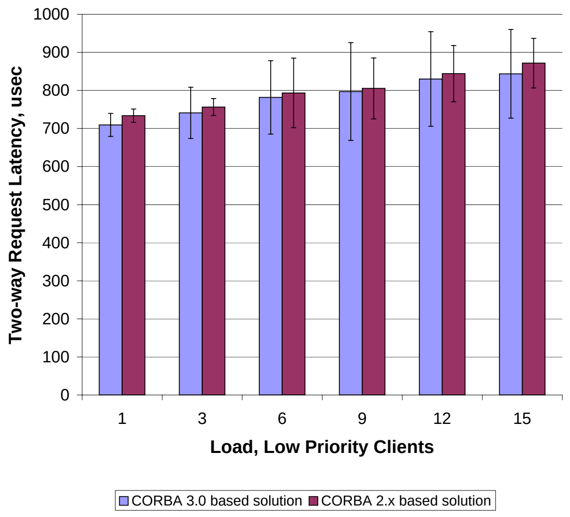

Figure 15: Latency for High-Priority Client Thread

In CORBA 2.4 configuration, the client sets the value of the *ClientPriority* policy to USE_THREAD_PRIORITY.

As was mentioned earlier, each low-priority thread has a different priority value, *e.g.*, 64, 65, etc. Such system configuration is more demanding than simply having all low-priority threads run at the same priority. By using different priority values for the low-priority threads, concurrency can be increased on the server. Thus, this design provides more opportunities for priority inversion to occur if the underlying ORB is not designed and implemented properly.

Results: Each solution was benchmarked with a different number of low-priority client threads: 1, 3, 6, 9, 12, and 15. For each solution and each number of client threads, the experiment was repeated three times. Figure 14 shows the jitter results. Figure 15 shows average latency (over three samples), with average jitter shown as error bars. The results in these figures illustrate that the CORBA 2.4-based latency is slightly lower than the CORBA 2.3 version, though its jitter is slightly higher. In general, therefore, the CORBA 2.4-based configuration provides a simpler programming model, smaller footprint, and faster initialization, with negligible impact on efficiency and predictability.

5.2 Buffered Request Benchmarks

Overview: We conducted benchmarks to compare the throughput of various buffered one-way request configurations. The *SyncScope* policy value was set to SYNC_NONE and the maximum message count of the buffer was increased for each test run.

Hardware/OS Benchmarking Platforms: We used two Dell computers, each with four Pentium Xeon CPUs running at 400 MHz and connected by a 100 bps Ethernet. The operating system of both machines was Windows NT 4.0. This OS, unlike most UNIX systems, does not limit the value of IOV_MAX, which is the maximum size of the vector used in *gather-write* operations. Thus, Windows NT is better equipped to demonstrate the full potential of performance gains from buffering one-way requests in the ORB. The timer used on both platforms was the high-resolution timer of the Intel chip, accessed by the Pentium RDTSC instruction.

Measurement techniques: A timestamp was obtained immediately before and after the execution of the request iteration loop. Throughput values were then calculated in calls/second for each test run of 2,000 requests. To closely emulate a real-time transport mechanism, such as VME, certain configuration policies were followed when conducting the one-way request benchmarks. In particular, Nagle's algorithm was disabled and the TCP window size was set to 8 kilobytes, which was the minimum supported on Windows NT. The size of each request was set to a larger value (9 kilobytes plus the header size) so that TCP would flow-control on each request and not buffer additional requests.

Results: In Figure 16, the throughput of unbuffered one-way requests is shown in the leftmost bar and is used as a baseline for comparison with the throughput of buffered one-way requests. To measure the effect of the buffer size, we set it to a different value in each test run. The other variables remain unchanged, *i.e.*, in all test runs the same number of messages is sent, and all the messages are identical. The figure shows that significant throughput gains (over 100% in the rightmost bar) can be realized by TAO's request buffering mechanism. However, tuning the queue size for each use-case is crucial to maximize performance.

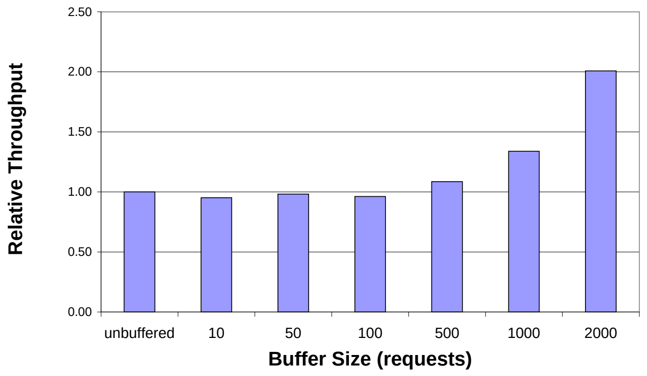

Figure 16: Throughput for Buffered One-way Requests

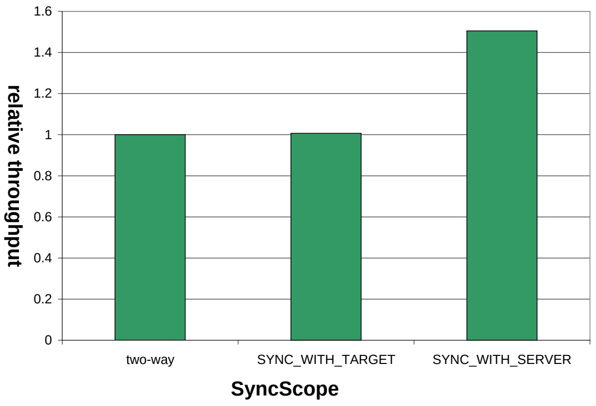

Figure 17: Throughput For Reliable One-way Requests

5.3 Reliable One-ways Benchmarks

Overview: We also conducted benchmarks to compare the performance of one-way requests configured with various levels of reliability. Test runs were made with one-way requests using the *SyncScope* policy values SYNC_WITH_SERVER and SYNC_WITH_TARGET. In addition, a test run was made using two-way requests via an operation call having the same signature as the one-way request used in the previous test runs. Throughput values were measured for each test run of 2,000 requests.

Measurement techniques: Timestamps, calculation of throughput, and TCP configuration policies for the reliable one-way benchmarks were the same as those for buffered one-way benchmarks. To simulate the work that an actual application would perform, the servant in this test executed 1,000 times a simple function that determines if an integer is a prime number. The hardware/software platform was identical to the one used for the buffered request benchmarks presented in Section 5.2.

Results: In Figure 17, the throughput of two-way requests is shown in the leftmost bar and is used as a baseline for comparison with the throughput of reliable one-way requests that use the *SyncScope* values SYNC_WITH_SERVER or SYNC_WITH_TARGET. This figure shows that the performance of two-way requests and one-way requests with the *SyncScope* policy value of SYNC_WITH_TARGET is almost identical. This result is expected because the processing of these two types of requests is identical on the server, and nearly identical on the client. Significant gains in throughput can be made, however, at the expense of complete end-to-end reliability, by setting the *SyncScope* policy value to SYNC_WITH_SERVER. The figure shows that the latter type of request obtains a throughput increase of more than 50%, compared with synchronous two-way requests.

6 Related Work

Real-time middleware is an emerging field of study. An increasing number of research efforts are focusing on integrating QoS and real-time scheduling into middleware like CORBA. Our previous work on TAO has examined many dimensions of ORB middleware design, including static [5] and dynamic [28] operation scheduling, event processing [6], I/O subsystem [31] and pluggable protocol [32] integration, synchronous [22] and asynchronous [21] ORB Core architectures, IDL compiler features [38] and optimizations [1], systematic benchmarking of multiple ORBs [39], patterns for ORB extensibility [14] and ORB performance [15]. In this section, we compare our work on TAO with related QoS middleware integration research.

6.1 CORBA-related QoS Research

URI TDMI: Wolfe *et al.* developed a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) [40]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed operation invocations (`TDMI`s) [41]. A `TDMI` corresponds to TAO's `RT_Operation` [5]. Likewise, an `RT_Environment` structure contains QoS parameters similar to those in TAO's `RT_Info`.

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

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

BBN QuO: The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [7]. QuO is based on CORBA and provides the following support for agile applications running in wide-area networks: (1) provides *run-time performance tuning and configuration* through the specification of operating regions, behavior alternatives, and reconfiguration strategies that allows the QuO run-time to adaptively trigger reconfiguration as system conditions change (represented by transitions between operating regions), (2) gives *feedback* across software and distribution boundaries based on a control loop in which client applications and server objects request levels of service and are notified of changes in service, and (3) supports *code mobility* that enables QuO to migrate object functionality into local address spaces in order to tune performance and to further support highly optimized adaptive reconfiguration.

The QuO model employs several *QoS definition languages* (QDLs) that describe the QoS characteristics of various objects, such as expected usage patterns, structural details of objects, and resource availability. QuO's QDLs are based on the separation of concerns advocated by Aspect-Oriented Programming (AoP) [42]. The QuO middleware adds significant value to adaptive real-time ORBs such as TAO. We are currently collaborating with the BBN QuO team to integrate the TAO and QuO middleware as part of the DARPA Quorum integration project.

UCSB Realize: The Realize project at UCSB [43] supports soft real-time resource management of CORBA distributed systems. Realize aims to reduce the difficulty of developing real-time systems and to permit distributed real-time programs to be programmed, tested, and debugged as easily as single sequential programs. Realize integrates distributed real-time scheduling with fault-tolerance, fault-tolerance with totally-ordered multicasting, and totally-ordered multicasting with distributed real-time scheduling, within the context of OO programming and existing standard operating systems. The Realize resource management model can be hosted on top of TAO [43].

UIUC Epiq: The Epiq project [44] defines a real-time CORBA mechanism that provides QoS guarantees and run-time scheduling flexibility. Epiq explicitly extends TAO's off-line scheduling model to provide on-line scheduling. In addition, Epiq allows clients to be added and removed dynamically via an admission test at run-time.

UCI TMO: The Time-triggered Message-triggered Objects (TMO) project [45] at the University of California, Irvine, supports the integrated design of distributed OO systems and real-time simulators of their operating environments. The TMO model provides structured timing semantics for distributed real-time object-oriented applications by extending conventional invocation semantics for object methods, *i.e.*, CORBA operations, to include (1) invocation of time-triggered operations based on system times and (2) invocation and time bounded execution of conventional message-triggered operations.

TAO differs from TMO in that it provides a complete CORBA ORB, as well as CORBA ORB services and real-time extensions. Timer-based invocation capabilities are provided through TAO's Real-Time Event Service [6, 9]. Where the TMO model creates new ORB services to provide its time-based invocation capabilities [13], TAO provides a subset of these capabilities by extending the standard CORBA COS Event Service. We believe TMO and TAO are complementary technologies because (1) TMO extends and generalizes TAO's existing time-based invocation capabilities and (2) TAO provides a configurable and dependable connection infrastructure needed by the TMO CNCM service. We are currently collaborating with the UCI TMO team to integrate the TAO and TMO middleware as part of the DARPA Quorum integration project.

6.2 Non-CORBA-related QoS Research

ARMADA: The ARMADA project [46, 47] defines a set of communication and middleware services that support fault-tolerant and end-to-end guarantees for real-time distributed applications. ARMADA provides real-time communication services based on the X-kernel and the Open Group's MK microkernel. This infrastructure provides a foundation for constructing higher-level real-time middleware services.

TAO differs from ARMADA in that most of the real-time infrastructure features in TAO are integrated into its ORB Core [22] and I/O subsystem [31], rather than in a microkernel. In addition, TAO implements the OMG CORBA standard, while also providing the hooks necessary to integrate with an underlying real-time I/O subsystem and OS. Thus, the real-time services provided by ARMADA's communication system can be utilized by TAO to support standards-based applications running over a vertically and horizontally integrated real-time system.

CMU Publisher/Subscriber: Rajkumar *et al.* [33] at CMU developed a real-time Publisher/Subscriber model that is similar to the TAO's Real-time Event Service [6], *e.g.*, it uses real-time threads to prevent priority inversion within its communication framework. The CMU model does not utilize any QoS specifications from publishers (event suppliers) or subscribers (event consumers), however. Therefore, scheduling is based on the assignment of request priorities, which is not addressed by the CMU model.

In contrast, TAO's Scheduling Service and real-time Event Service utilize QoS parameters from suppliers and consumers to assure resource access via priorities. One interesting aspect of the CMU Publisher/Subscriber model is the separation of priorities for subscription and data transfer. By handling these activities with different threads, with possibly different priorities, the impact of on-line scheduling on real-time processing can be minimized.

7 Concluding Remarks

Real-time distributed object computing (DOC) middleware is a promising solution for key challenges facing researchers and developers of real-time applications. Designing and optimizing standards-based and commodity-off-the-shelf (COTS) DOC middleware that can meet the QoS requirements of real-time applications requires an integrated architecture that can deliver end-to-end QoS support at multiple levels in real-time and embedded systems. The Real-time CORBA [18] and Messaging [19] specifications in CORBA 2.4 are an important step in this direction.

Unfortunately, the CORBA 2.4 specification lacks sufficient specificity to *portably* manage processor, communication, and memory resources for applications with stringent QoS requirements. For example, Section 4.2.2 describes outlines a number of problems with the standard Real-time CORBA *priority mapping*, *priority model*, *thread pool*, and *priority banded connection* mechanisms intended to preserve request priorities end-to-end. The following discussion summarizes our lessons learned developing and deploying an implementation of CORBA and representative applications that are based on the CORBA 2.4 specifications.

Control over thread priority must be end-to-end: Ensuring appropriate end-to-end QoS for real-time applications requires more than just the implementation of the Real-time CORBA specification and the mapping of global priorities between ORBs. It also requires control over end-to-end thread priorities, connection $\leftrightarrow$ thread pool associations, and other ORB endsystem resources used to process a request. Moreover, applications must be able to control these resources at each layer of an DOC middleware. The Real-time CORBA specification [18] adopted for CORBA 2.4 allows implementations to provide end-to-end guarantees, but it does not require them. Moreover, it does not provide explicit mechanisms to control how I/O threads in an ORB Core map to thread priorities and connections.

Section 4.2.2 describes the policies and mechanisms we used in TAO's prioritized connection endpoints extension to precisely define these associations and ORB configurations. Application developers can use this extension to ensure that the client and server ORBs process the request at the appropriate priority end-to-end. In previous work [31], we described how TAO can be integrated with an *early demultiplexing* feature in the operating system's I/O subsystem to ensure that the OS kernel and network interfaces also preserve the priority of the request.

One-way semantics must be specified precisely: The CORBA 2.3 specification defines the semantics of one-way operations to receive best-effort service. However, it provides no further end-to-end delivery guarantees to applications. Thus, a CORBA 2.3-compliant ORB can simply drop every request, deliver it at some arbitrary future time (but out of order with subsequent requests), or send it immediately. The CORBA 2.4 Messaging specification provides better control over the semantics of oneway operations, but it falls short in the SYNC_NONE case, where it does not provide mechanisms to control buffering limits and/or flushing policies. Section 4.2.4 describe the policies and mechanisms we used in TAO to define one-way buffering semantics precisely and ensure adequate throughput for certain types of real-time applications.

Standards-based COTS DOC middleware solutions must be usable: COTS DOC middleware is motivated by the need to reduce maintenance and development costs, as well as improve time-to-market cycles. These same forces also make overly complex or non-scalable DOC middleware solutions undesirable. Thus, COTS DOC middleware must not only provide a standardized programming model, but it must also define precise semantics to meet stringent end-to-end QoS requirements *and* implement these semantics via a convenient API. Our experience developing standards-compliant CORBA middleware shows that it is possible to achieve both these goals, although the CORBA 2.4 specification can be improved by applying the TAO enhancements described in this paper.

The open-source code, benchmarks, and documentation for TAO is freely available and can be downloaded from URL `www.cs.wustl.edu/~schmidt/TAO.html`. Our focus on the TAO project has been to research, develop, and optimize policies and mechanisms that allow CORBA to support applications with hard real-time requirements. These requirements motivate many of the optimizations and design strategies presented in this paper.

TAO has been used on a wide range of distributed real-time and embedded systems, including an avionics mission computing architecture for Boeing [6], the next-generation Run Time Infrastructure (RTI) implementation for the Defense Modeling and Simulation Organization's (DMSO) High Level Architecture (HLA) [48], and high-energy physics experiments at SLAC [49] and CERN [50].

References

[1] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.

[3] Sun Microsystems, Inc, *Java Remote Method Invocation Specification (RMI)*, Oct. 1998.

[4] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[5] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[6] 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.

[7] 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.

[8] 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.

[9] C. O'Ryan, D. C. Schmidt, and D. Levine, "Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations," in *Proceedings of the 5^{th} Workshop on Object-oriented Real-time Dependable Systems*, (Montery, CA), IEEE, Nov. 1999.

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

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

[12] DARPA, "The Quorum Program." http://www.darpa.mil/ito/research/quorum/index.html, 1999.

[13] K. Kim and E. Shokri, "Two CORBA Services Enabling TMO Network Programming," in *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*, IEEE, January 1999.

[14] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*. New York, NY: Wiley & Sons, 2000.

[15] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazine*, vol. 8, no. 1, 2000.

[16] M. Fayad, R. Johnson, and D. C. Schmidt, eds., *Object-Oriented Application Frameworks: Problems & Perspectives*. New York, NY: Wiley & Sons, 1999.

[17] 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.

[18] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

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

[20] R. Callison, M. Goo, and D. Butler, "Real-time CORBA Trade Study," Tech. Rep. D204-31159, The Boeing Company, 1999.

[21] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[22] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2001.

[23] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[24] 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.

[25] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[26] D. C. Schmidt and F. Kuhns, "An Overview of the Real-time CORBA Specification," *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing*, June 2000.

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

[28] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, To appear 2001.

[29] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[30] D. L. Levine, D. C. Schmidt, and S. Flores-Gaitan, "An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers," in *Proceedings of Multimedia Computing and Networking 2000 (MMCN00)*, (San Jose, CA), ACM, Jan. 2000.

[31] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, To appear 2001.

[32] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

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

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

[35] R. Lachenmaier, "Open Systems Architecture Puts Six Bombs on Target." www.cs.wustl.edu/$\sim$schmidt/TAO-boeing.html, Dec. 1998.

[36] M. de Sousa, "Mapping Synchronisation Protocols onto Real-Time CORBA," in *Proceedings of the 2^{nd} International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Sept. 2000.

[37] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[38] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, and M. Kircher, "Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks," *C++ Report*, vol. 12, Mar. 2000.

[39] 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.

[40] 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.

[41] 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.

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

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

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

[45] K. H. K. Kim, “Object Structures for Real-Time Systems and Simulators,” *IEEE Computer*, pp. 62–70, Aug. 1997.

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

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

[48] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems*. Upper Saddle River, New Jersey: Prentice Hall PTR, 1999.

[49] SLAC, “BaBar Collaboration Home Page.” http://www.slac.stanford.edu/BFROOT/.

[50] A. Kruse, “CMS Online Event Filtering,” in *Computing in High-energy Physics (CHEP 97)*, (Berlin, Germany), Apr. 1997.