Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0

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 proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium (RTAS'00), Washington D.C., USA, 31 May – 2 June 2000.

Abstract

To be an effective platform for performance-sensitive real-time systems, commercial-off-the-shelf (COTS) distributed object computing (DOC) middleware must support application quality of service (QoS) requirements end-to-end. However, conventional 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 forthcoming CORBA 3.0 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. However, these specifications still lack sufficient detail 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 3.0 Real-time and Messaging specifications provide a starting point for addressing the needs of an important class of applications with stringent real-time requirements. Second, we illustrate how the CORBA 3.0 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 empirically evaluate the performance of TAO to illustrate how its features address the QoS requirements of certain types of real-time applications.

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

*This work was funded in part by 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 [1]. 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 and quality.

Requirements for increased software productivity and quality motivate the use of distributed object computing (DOC) *middleware* [2]. 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 [3]. The *business goal* of DOC middleware is to decrease the cycle-time and effort required to develop high-quality 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 [1] have impeded the development and deployment of efficient, predictable, and scalable middleware for real-time systems. In particular, 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 [4]. 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 → 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 [4]. Over the past two years, however, the use of CORBA-based [5] 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 largely from the following factors:

1. The maturation of standards: During the past decade, the OMG's suite of CORBA standards has matured considerably, particularly with respect to the specification of components and features for high-performance and real-time systems. For instance, the forthcoming CORBA 3.0 standard[1] includes Messaging [12] and Real-time CORBA specifications [11]. The Messaging specification defines asynchronous operation models [14] and allows applications to control many end-to-end ORB QoS policies. The Real-time CORBA specification defines interfaces and policies for managing ORB processing, communication, and memory resources. Figure 1 illustrates how these various CORBA 3.0 features interact.

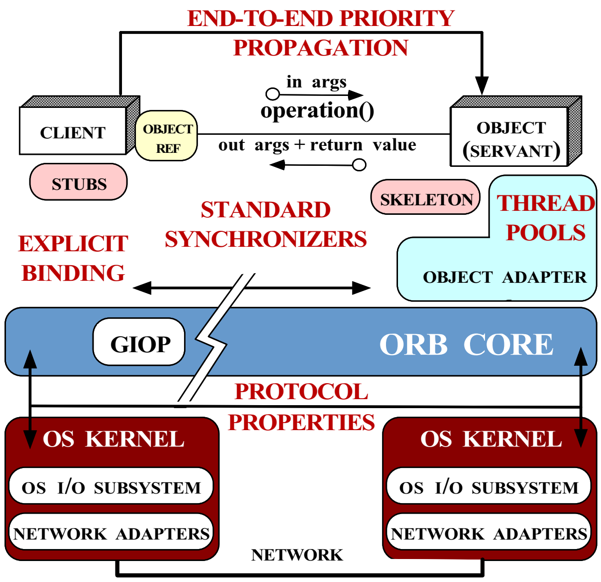

Figure 1: CORBA 3.0 Support for Real-Time Applications

[1] The complete CORBA 3.0 standard will be available in late 2000. However, many key components [10, 11, 12, 13] are already specified.

2. The maturation of patterns and frameworks: In recent years, a substantial amount of R&D effort has focused on patterns and frameworks for high-performance and real-time systems. For instance, research conducted as part of the DARPA Quorum project [15, 7, 4, 16] has identified key design patterns [17] and optimization principle patterns [18]. Moreover, these projects have instantiated these patterns to create high-quality frameworks [19, 20, 21] for QoS-enabled DOC middleware and real-time applications. An increasing number of COTS ORBs [22] are applying these patterns and frameworks to implement the CORBA Real-time [4] and Messaging [14] specifications.

The vehicle for our research on DOC middleware for high-performance and real-time applications is TAO [4], which 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, connection [23] and demultiplexing [18] patterns [2, 17]. 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 [11] and Messaging [12] 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 3.0 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 standard CORBA 3.0 features as well as TAO extensions, and describes how they address the needs of an avionics mission computing system, a representative of a class of hard real-time applications;

Section 3 evaluates the results of benchmarks that measure efficiency, predictability, and scalability of TAO's real-time CORBA features; and Section 4 presents concluding remarks.

2 Evaluating CORBA 3.0 Features and Extensions

2.1 Overview

During the past three years, we developed two versions of a real-time avionics mission computing application, each using a different version of TAO. The first version, described in [6], is based on CORBA 2.x and uses multiple instances of the

ORB, one for each priority level. This approach results in minimal priority inversion, as no resources are shared between different priority levels. However, it is hard to program to, since the servers must activate each servant multiple times, once under each ORB, and the client must manually select the right object reference for each priority level.

In the following sections we describe real-time policies and mechanisms used in our second implementation, where TAO is based on the Real-time [11] and Messaging [12] specifications. We also motivate several enhancements we developed to provide more precise control over CORBA 3.0 Real-time and Messaging features.

2.2 Preserving Priorities End-to-End

Systems with stringent QoS requirements, such as our avionics mission computing application, often must execute a request at the same priority, end-to-end. In the following paragraphs we discuss the mechanisms provided by the Real-time CORBA [11] specification to satisfy this requirement. These mechanisms are: priority mapping, priority models, thread pools and priority banded connections.

Priority Mapping: The specification defines a universal, platform independent priority scheme called *Real-time CORBA Priority*. Real-time CORBA Priority 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 *Real-time CORBA priority* into *native OS priority*. The specification mandates each ORB to provide default priority mapping functions as well as a mechanism to allow users to override these defaults. However, the specification does not state how those mappings functions are accessed and set, which forces the application developer to use proprietary interfaces (this omission will hopefully be rectified in future versions of the specification).

CORBA Priority models: The Real-time CORBA specification defines a *PriorityModel* policy which is used to determine 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 2 (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. This model is not appropriate for applications, such as our avionics mission computing scenario, that invoke the *same* operation on the *same* object, but at *different* priorities from *different* client threads.

If a mission computing application were to use the SERVER_DECLARED priority model, it would have to acti-

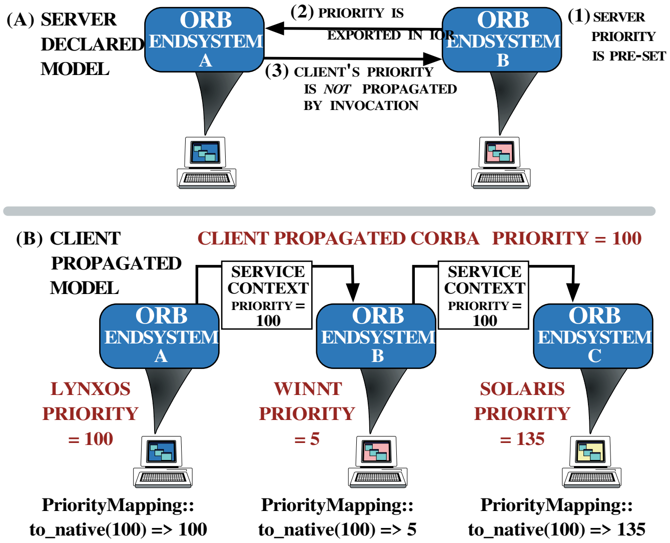

Figure 2: Real-time CORBA Priority Models

vate 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. This solution would be unnecessarily complicated, however, and would interact poorly with CORBA location services, such as Naming or Trading, because each object would have to be registered multiple times. Moreover, 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.x-based implementation, we faced similar challenges in managing multiple object references corresponding to multiple server ORBs. Our experience indicates that such approaches yield complex code that is hard to maintain, thereby negating several advantages of COTS middleware.

In the other priority model offered by the Real-time CORBA specification, the CLIENT_PROPAGATED model shown in Figure 2 (B), the server honors the priority of the invocation set by the client. In this model, the client encapsulates its priority in the service context list of the operation invocation. When a server parses the request, it extracts the priority from the service context and sets the priority of the processing thread to match the requested priority. However, this model creates opportunities for priority inversion, because it is possible that the initial priority of the thread reading the request is too high or too low.

Thread pools: Thread pools are another mechanism defined by the Real-time CORBA specification that initially appear to resolve the limitations with the two standard priority models outlined above. A Real-time CORBA server can associate each POA[2] with a pool of pre-allocated threads running at the appropriate priorities. However, the specification does not require that threads in a pool receive requests directly from connections. In fact, a compliant implementation may choose to have 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 3. Such an implementation can increase av-

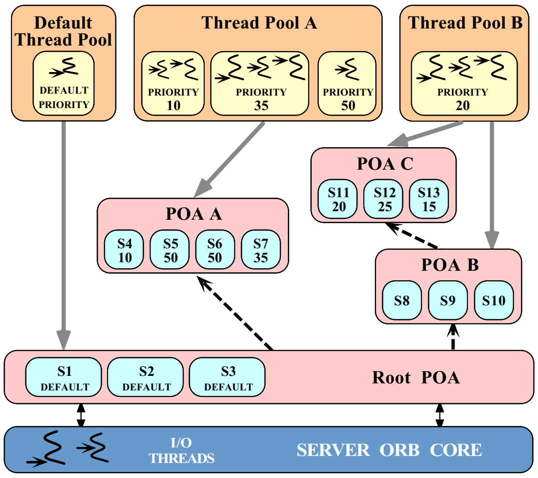

Figure 3: POA Thread Pools in Real-time CORBA

erage and worst-case latency and create opportunities for unbounded priority inversions [18], 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.

A pool can be optionally pre-configured for a maximum buffer size or number of requests, as shown in Figure 4. If

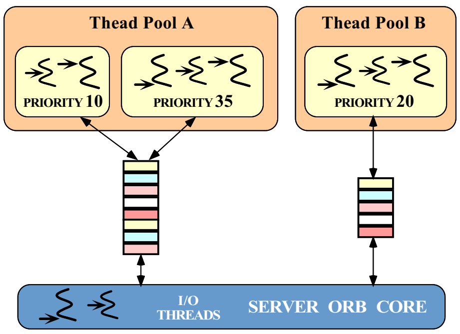

Figure 4: Buffering Requests in RT-CORBA Thread Pools

buffering is enabled for the pool, the request will 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: The *priority banded connections* feature defines yet another way to overcome the limitations of the two standard priority models. This facility allows a client to communicate with the server via multiple transport connections - each dedicated to carrying invocations of distinct CORBA priority or range of priorities, as shown in Figure 5. A client ORB establishes a priority banded con-

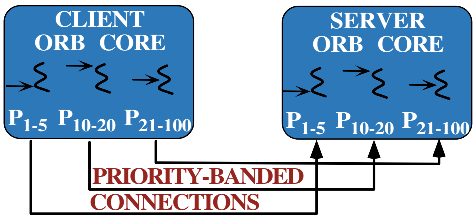

Figure 5: Priority Banded Connections

nection by sending a server the `_bind_priority_band` request, which specifies the range of priorities the connection will be used for. This allows the server to allocate the necessary resources for the connection and to configure these resources to provide service for the specified priority range. 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.

Unfortunately, the CORBA 3.0 Real-time specification does *not* define how an ORB should handle the following issues:

- **Server connections ↔ thread pool associations:** There is no standard API 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.

- **Server connections ↔ thread priority associations:** 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 suf-

[2]A POA (Portable Object Adapter) is a server side entity responsible for creating object references, activating objects, and dispatching requests made on objects to their respective servants.

fer from problems similar to those with POA thread pools, as shown in Figure 3.

As described above, the mechanisms defined in the Real-time CORBA portion of the CORBA 3.0 specification provide application developers with greater control over ORB endsystem resources than earlier CORBA 2.x specifications. For many real-time applications these mechanisms are sufficient to provide the necessary QoS guarantees. For real-time applications with stringent QoS requirements however, this lack of specificity can lead to ineffective or *non-portable* implementations. Therefore, the solution we applied in TAO defines mechanisms for explicitly mapping thread pools and thread priorities to connection endpoints.

Implementing prioritized connection endpoints in TAO: To alleviate the problems listed above, we extended the standard Real-time CORBA mechanisms to give TAO applications greater control over the mapping of connections to thread priorities within the ORB Core. Below, we describe key mechanisms provided by TAO to implement this feature in client and server ORBs:

• **Server ORB support for binding thread priorities to listen-mode connection endpoints:** TAO allows server to have multiple listen-mode [24] 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 allows a client to receive service of the 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 for sending the request to the server. The profile is selected based on the priority at which the client wants to have 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 we want to preserve the priority of request end-to-end.
- USE_PRIORITY_RANGE – 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 basic priority models and the priority banded connections mechanism defined by the Real-time CORBA specification to achieve an effective balance between the SERVER_DECLARED and CLIENT_PROPAGATED models. In particular, they provide the same degree of control to the server as the SERVER_DECLARED model by restricting clients to use well-known priorities. However, they allow clients to select a priority that best meets their requirements from priorities published by the server. It is important to note that while this balance is achieved, priority inversions are avoided, programming model remains simple, and the resources are strictly controlled.

Figure 6 provides an example of TAO client and server using prioritized connection endpoints mechanism. The server is an Event Channel consumer application [25], with four connection endpoints, EP_1 to 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 6 (A). The client, which is an Event Channel supplier application, has four threads with priorities $P1$ to $P4$. When a supplier thread makes a call on the consumer object reference exported by the server, the *ClientPriority* policy is checked before the client ORB selects a profile from the object reference. Assume that the *ClientPriority* policy value is set to USE_THREAD_PRIORITY. In that case, when a client thread with priority $P1$ invokes an operation on

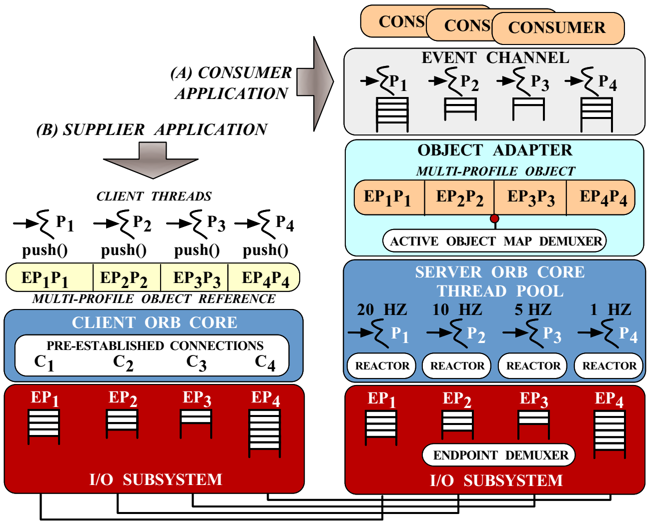

Figure 6: TAO's New CORBA 3.0-based Solution

a consumer object, the ORB selects the profile corresponding to that priority, *i.e.* the one that contains connection endpoint EP_1. The ORB then 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.

In summary, application developers can use the prioritized connection endpoint extension that TAO adds to the Real-time CORBA specification's priority banded connections model to ensure that the client and server ORBs process the request at the appropriate priority end-to-end. In previous work [26], 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.

Section 3.1 illustrates the performance of TAO's prioritized connection endpoint architecture.

2.3 Ensuring Adequate Operation Throughput

Distributed real-time applications often have stringent timing requirements: the 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 typically 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. The following two problems can arise, however, 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 temporarily unavailable. However, ORB transport protocols, such as IIOP, often implement reliable data delivery using a sliding window flow control algorithm [24]. Therefore, 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. However, buffering at the application-level increases the burden on application developers, 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 the footprint of the application.

Often, a better solution is for the ORB to buffer one-way and asynchronous invocations transparently.[3] At some later time, the buffered requests can be delivered *en masse* to the server. By buffering requests, a client ORB amortizes the per-operation processing overhead and increases effective network utilization. Moreover, the ORB can use OS *gather write* operations, such as `writev` [24], to minimize the number of mode switches needed to transmit the buffered requests. In addition, 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 buffered requests in TAO: The CORBA 3.0 Messaging specification introduces several mechanisms to give application developers more control over QoS parameters than in CORBA 2.x specifications. In particular, applications

[3] 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.

can use CORBA 3.0 features, such as the *SyncScope* policy, to control latency/reliability tradeoffs. For example, applications can use the SYNC_WITH_SERVER policy 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, the CORBA Messaging specification provides no 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. When user-specified buffering limits are reached, the buffers are flushed and the queued requests are delivered to the server.

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 a user specified high-water mark, the buffered requests are delivered to the server. This approach allows the user to batch *n* requests together.

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

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

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

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

Section 3.2 presents benchmarks illustrating the performance of TAO's buffered requests implementation.

2.4 Evaluation

Our CORBA 3.0-based avionics mission computing solution has the following improvements over the original CORBA 2.x-based design:

More standard programming model: The CORBA 3.0 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, the application programming model is simplified and the portability of the software increases because the system is based on a standard.

Reduced memory footprint: Our original solution required that multiple ORBs be created within each client and server, once for each rate group. Moreover, servants would have multiple object references, one for each global priority. This resulted in a relatively large application footprint. In contrast, our CORBA 3.0-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 3.0-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 [26] 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 3.0 Messaging specification [12] gives clients greater control over these tradeoffs.

In addition to alleviating the drawbacks with our original CORBA 2.x-based solution, the new CORBA 3.0-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.x-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 3.0-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.x-based approach may require modification to the service itself. For example, the Event Service invokes operations on application-provided objects. In order to make these invocations 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. On the other hand, in the CORBA 3.0-based approach all these tasks are performed by the ORB, transparently to applications and services.[4]

Easier integration with real-time scheduling services: The CORBA 3.0 Real-time specification supports higher-level CORBA scheduling services that allocate resources end-to-end. For example, TAO's static scheduling service [4] can associate application activities[5] with global CORBA priorities. Such scheduling services can use the priority transformations and the policy framework defined in the CORBA Messaging [12] to create sophisticated and adaptive real-time applications.

3 The Performance of the TAO Real-time CORBA ORB

3.1 Preserving Priorities End-to-End

Overview: The benchmarks in this section compare the performance of TAO's CORBA 2.x-based solution, *i.e.*, using an ORB-per-priority architecture, with TAO's CORBA 3.0-based solution, *i.e.*, using the prioritized connection mechanism described in Section 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.

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

[5] 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.

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.x-based solution and CORBA 3.0-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.x-based approach, this is achieved by creating an ORB-per-thread for each test priority. In the CORBA 3.0-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.x 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. In CORBA 3.0 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 7 shows the jitter

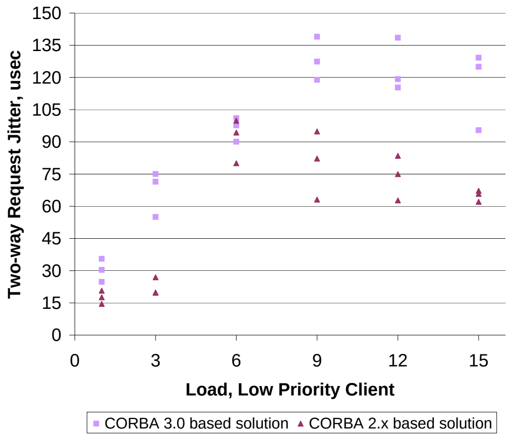

Figure 7: Jitter for High-Priority Client Thread

results. Figure 8 shows average latency (over three samples), with average jitter shown as error bars. The results in these

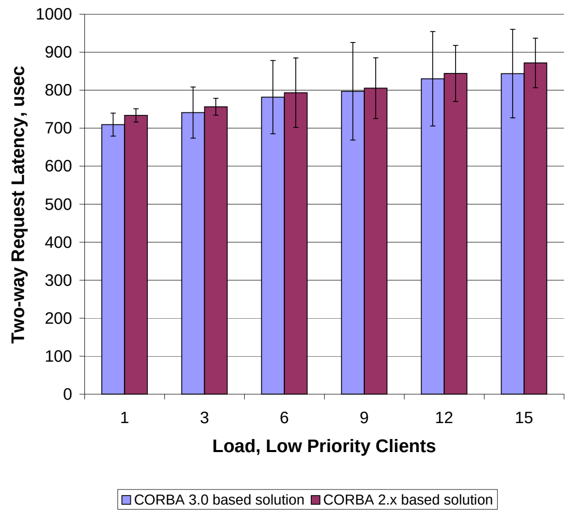

Figure 8: Latency for High-Priority Client Thread

figures illustrate that the CORBA 3.0-based latency is slightly lower than the CORBA 2.x version, though its jitter is slightly higher. In general, therefore, the CORBA 3.0-based configuration provides a simpler programming model, smaller footprint, and faster initialization, with negligible impact on efficiency and predictability.

3.2 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 9, 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

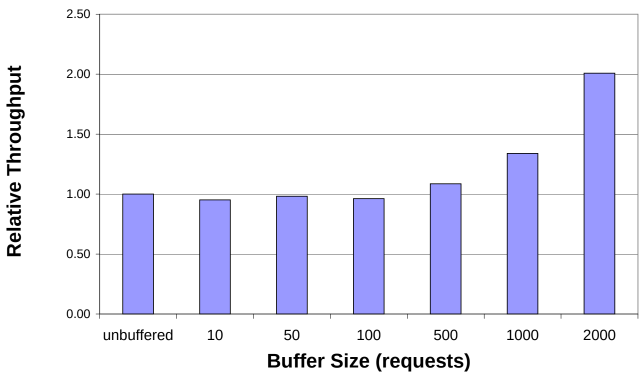

Figure 9: Throughput For Buffered One-way Requests

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 achieve maximum performance.

4 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 commercial-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 OMG CORBA 3.0 Real-time CORBA [11] and Messaging [12] specifications are an important step in this direction. However, they lack sufficient specificity to *portably* manage processor, communication, and memory resources for applications with stringent QoS requirements. The following discussion summarizes our lessons learned from developing and deploying an implementation of CORBA that is based on the CORBA 3.0 specifications.

Control over thread priority must be end-to-end: Ensuring appropriate end-to-end QoS for real-time applications involves 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 [11] allows implementations to provide end-to-end guarantees, but it does not require them. Nor does it provide explicit mechanisms to control how I/O threads in an ORB Core map to thread priorities and connections. Section 2.2 describes the policies and mechanisms we used in TAO's prioritized connection endpoints extension to precisely define these associations and ORB configurations.

One-way semantics must be specified precisely: The CORBA 2.x 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.x-compliant ORB could simply drop every request, deliver it at some arbitrary future time (but out of order with subsequent requests), or send it immediately. The CORBA 3.0 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 2.3 describe the policies and mechanisms we used in TAO to precisely define one-way buffering semantics and ensure adequate throughput for certain types of real-time applications.

The open-source code, benchmarks, and documentation for TAO is freely available and can be downloaded from URL `http://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. 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) [27], and high-energy physics experiments at SLAC [28] and CERN [29].

References

[1] C. D. Gill, F. Kuhns, D. L. Levine, D. C. Schmidt, B. S. Doerr, R. E. Schantz, and A. K. Atlas, "Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems," in *Proceedings of the 1st IEEE International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, Nov. 1999.

[2] 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.

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

[4] 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.

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

[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] S. Vinoski, "New Features for CORBA 3.0," *Communications of the ACM*, vol. 41, pp. 44–52, October 1998.

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

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

[13] BEA Systems, *et al.*, *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.

[14] 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.

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

[16] 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.

[17] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.

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

[19] 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.

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

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

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

[23] 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 2000.

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

[25] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Performance of TAO's Pluggable Protocols Framework on High-speed Embedded Interconnects," Department of Computer Science, Technical Report WUCS-99-12, Washington University, St. Louis, 1999.

[26] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

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

[28] SLAC, "BaBar Collaboration Home Page." http://www.slac.stanford.edu/BFROOT/.

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