Techniques for Enhancing Real-time CORBA Quality of Service [*]

Irfan Pyarali[†]
irfan@oomworks.com
OOMWorks, LLC
Metuchen, NJ

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer Engineering
University of California, Irvine, CA

Ron K. Cytron
cytron@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO

Abstract

End-to-end predictability of remote operations is essential for many fixed-priority distributed real-time and embedded (DRE) applications, such as command and control systems, manufacturing process control systems, large-scale distributed interactive simulations, and testbeam data acquisition systems. To enhance predictability, the Real-time CORBA specification defines standard middleware features that allow applications to allocate, schedule, and control key CPU, memory, and networking resources necessary to ensure end-to-end quality of service support.

This paper provides two contributions to the study of Real-time CORBA middleware for DRE applications. First, we identify potential problems with ensuring predictable behavior in conventional middleware by examining the end-to-end critical code path of a remote invocation and identifying sources of unbounded priority inversions. Experimental results then illustrate how the problems we identify can yield unpredictable behavior in conventional middleware platforms. Second, we present design techniques for ensuring real-time QoS in middleware. We show how middleware can be redesigned to use non-multiplexed resources to eliminate sources of unbounded priority inversion. The empirical results in this paper are conducted using TAO, which is widely-used and open-source DRE middleware compliant with the Real-time CORBA specification.

1 Introduction

The maturation of the CORBA specification [1] and CORBA-compliant object request broker (ORB) implementations have simplified the development of distributed applications with complex *functional* requirements. Next-generation distributed real-time and embedded (DRE) applications, however, also have complex *quality of service* (QoS) requirements, such as stringent bandwidth, latency, jitter, and dependability needs. These needs have not been well served by early versions of CORBA due to the lack of QoS support. The integration of Real-time CORBA 1.0 [2] into the CORBA specification has therefore been an important step towards the creation of standard, commercial-off-the-shelf (COTS) middleware that can deliver end-to-end QoS support in DRE systems.

As shown in Figure 1, a Real-time CORBA ORB endsystem consists of network interfaces, an I/O subsystem, other OS mechanisms, and ORB middleware capabilities that support end-to-end predictability for operations in *fixed-priority* CORBA applications. These capabilities [3] include standard interfaces and QoS policies that allow applications to configure and control the following resources:

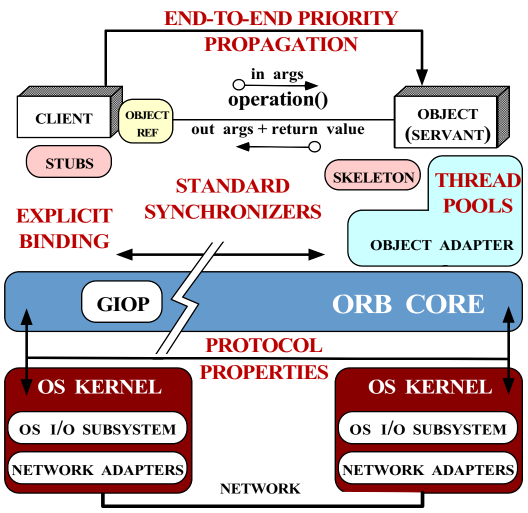


Figure 1: **Standard Capabilities in Real-time CORBA ORB Endsystems**

- *Processor resources* via thread pools, priority mechanisms, and intraprocess mutexes
- *Communication resources* via protocol properties and explicit bindings with non-multiplexed connections and
- *Memory resources* via buffering requests in queues and bounding the size of thread pools.

Our prior work on Real-time CORBA has explored many dimensions of ORB design and performance, including scalable event processing [4], request demultiplexing [5], I/O subsystem [6] and protocol [7] integration, connection management [8] and explicit binding [9] architectures, asyn-

[*]This work was supported in part by AFOSR grant F49620-00-1-0330, Boeing, DARPA ITO contract F33615-00-C-1697 (PCES) and AFRL contracts F3615-97-D-1155/DO (WSOA) and F33645-97-D-1155 (ASTD/ASFD).

[†]Work done by the author while at Washington University.

chronous [10] and synchronous [11] concurrent request processing, and IDL stub/skeleton optimizations [12]. In this paper, we consider how to achieve *end-to-end predictability* using Real-time CORBA. We first describe the end-to-end critical code path of remote CORBA invocations to identify sources of unbounded priority inversions. We then present experimental results that show how the identified problems can yield non-predictable behavior in conventional ORB middleware platforms and how Real-time CORBA features can be applied to alleviate these problems.

The vehicle for our work is TAO [13], which is a high-quality, widely-used, open-source[1] ORB compliant with most of the CORBA 2.6 specification [1]. Although TAO-specific capabilities have been used in mission-critical DRE applications for the past five years [14], we only recently finished enhancing TAO to support the Real-time CORBA 1.0 specification. TAO now implements the standard Real-time CORBA APIs that are designed to meet the real-time requirements of fixed priority applications by respecting and propagating thread priorities, avoiding unbounded priority inversions, and allowing applications to configure and control processor, communication, and memory resources.

The remainder of this paper is organized as follows: Section 2 examines the design of TAO's Real-time CORBA ORB, focusing on how it eliminates key sources of unbounded priority inversion; Section 3 describes the Real-time CORBA testbed and experiments used to evaluate TAO's end-to-end performance; Section 4 examines the experimental results that demonstrate the end-to-end predictability of TAO; Section 5 compares our research on TAO with related work; and Section 6 presents concluding remarks.

2 Redesigning TAO to Achieve End-to-End Predictability

This section examines the design of two versions of TAO—before and after Real-time CORBA support was added—and shows how we identified and eliminated sources of unbounded priority inversion in TAO. To identify sources of unbounded priority inversion, we first analyze the typical end-to-end critical code path of a CORBA request within ORBs that implement the "classic CORBA" specification (*i.e.*, before Real-time CORBA was adopted).[2] We then show how non-multiplexed resources can be used to eliminate key sources of unbounded priority inversion.

[1]The source code, documentation, and performance tests for TAO can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

[2]The Real-time CORBA 1.0 specification was officially integrated into the baseline CORBA specification in version 2.4 [15].

2.1 Tracing a CORBA Two-Way Invocation End-to-End

To identify sources of unbounded priority inversion within an ORB, we analyze the end-to-end critical code path of a synchronous two-way CORBA request, *e.g.*,

```
result = object→operation (arg1, arg2)
```

The numbered bullets below correspond to the steps illustrated in Figure 2. Although we describe these steps in terms of a version of TAO that implemented only classic CORBA, other implementations of classic CORBA behave similarly. Moreover, we generalize the discussion by describing the steps in terms of the Reactor, Acceptor-Connector, and Leader/Followers patterns described in Sidebar 1.

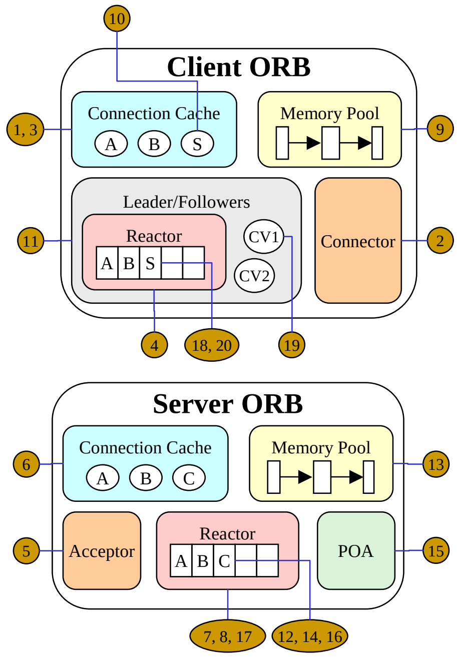


Figure 2: **Tracing an Invocation Through a Classic CORBA ORB**

Connection management. We first describe how a connection can be established between a CORBA client and server. The following are the activities a client ORB performs to create a connection actively when a client application invokes an operation on an object reference to a target server object:

1. Query the client ORB's connection cache for an existing connection to the server designated in the object reference on which the operation is invoked.
2. If the cache doesn't contain a connection to the server, use a connector factory [16] to create a new connection S.
3. Add the newly established connection S to the connection cache.

Sidebar 1: Network Programming Patterns in TAO

TAO's software architecture—like the architecture of many CORBA ORBs—is based on the network programming patterns described in [16]. Three fundamental patterns used in TAO's ORB are outlined below:

- **Reactor** is an event handling design pattern that allows an ORB to demultiplex and dispatch incoming requests and replies from peer ORBs. Request and reply handlers register with a reactor, which uses a synchronous event demultiplexer to wait for data to arrive from one or more ORBs. When data arrives, the synchronous event demultiplexer notifies the reactor, which then synchronously dispatches the handler associated with the data so that it can perform the requested service.
- **Acceptor-Connector** is an initialization design pattern that decouples the connection establishment between ORBs from the processing they perform after they are connected. The ORB initiating the connection uses a *connector factory* to establish the connection actively. The receptor ORB uses an *acceptor factory* to establish the connection passively.
- **Leader/Followers** is an architectural design pattern that provides an efficient concurrency model where multiple threads take turns detecting, demultiplexing, dispatching, and processing requests and replies from peer ORBs.

4. Also add connection S to the client ORB's reactor [16] since S is bi-directional and the server may send requests to the client using S.

The server ORB activities for accepting a connection passively are described next:

5. Use an acceptor factory [16] to accept the new connection C from the client.
6. Add C to the server ORB's connection cache since C is bi-directional and the server can use it to send requests to the client.
7. Also add connection C to the server ORB's reactor so the server is notified when a request arrives from the client.
8. Wait in the reactor's event loop for new connection and data events.

Synchronous two-way request/reply processing. We now describe the steps involved when a client invokes a synchronous two-way request to a server, the server processes the request and sends a reply, and the client processes the reply. After the connection from client to server has been established, the following activities are performed by a client ORB when a client application thread invokes an operation on an object reference that designates a target server object:

9. Allocate a buffer from a memory pool to marshal the parameters in the operation invocation.
10. Send the marshaled data to the server using connection S. Connection S is locked for the duration of the transfer.
11. Use the leader/followers manager [16] to wait for a reply from the server. Assuming that a leader thread is already available, the client thread waits as a follower on a condition variable or semaphore.[3]

The server ORB activities for processing a request are described below:

12. Read the header of the request arriving on connection C to determine the size of the request.
13. Allocate a buffer from a memory pool to hold the request.
14. Read the request data into the buffer.
15. Demultiplex the request to find the target portable object adapter (POA) [17], servant, and skeleton – then dispatch the designated upcall to the servant after demarshaling the request parameters.
16. Send the reply (if any) to the client on connection C. Connection C is locked for the duration of the transfer.
17. Wait in the reactor's event loop for new connection and data events.

Finally, the client ORB performs the following activities to process a reply from the server:

18. The leader thread reads the reply from the server on connection S.
19. After identifying that the reply belongs to the follower thread, the leader thread hands off the reply to the follower thread by signaling the condition variable used by the follower thread.
20. The follower thread demarshals the parameters and returns control to the client application, which processes the reply.

2.2 Identifying Sources of Unbounded Priority Inversion

Devising predictable components and their interconnections is essential to provide end-to-end QoS support for ORB endsystems. This section identifies sources of unbounded priority inversion that often arise in the critical path of a synchronous CORBA two-way operation invocation outlined in Section 2.1. The steps we refer to in our discussion below appear in Figure 2.

Connection Cache. In steps 10 and 16, other threads are denied access to the connection for the duration of the data transfer. This connection-level mutual exclusion prevents multiple threads from writing to the same connection simultaneously

[3]The leader thread may actually be a server thread waiting for incoming requests or another client thread waiting for its reply.

and corrupting request and reply data. However, the time required to send the request data depends on the availability of network resources and the size of the request. Unless the underlying network provides strict time guarantees for data delivery, serializing access to a connection can cause a higher priority thread to wait indefinitely, thereby yielding unbounded priority inversion. Moreover, if priority inheritance [18] is not supported by the mutual exclusion mechanism that serializes the connection, priority inversion can be further exacerbated.

An ORB can create a new connection to the peer ORB instead of waiting for existing connection to become available. However, creating a new connection could also take an indefinite amount of time and therefore would not bound the priority inversion. Another approach would be to preallocate enough connections to ensure the client always has a non-busy connection available to the server. Unfortunately, this scheme requires advanced knowledge of application behavior, is resource intensive, and scales poorly.

Memory freestore. In steps 9 and 13, buffers are allocated to marshal and demarshal requests. While finding a sufficiently large buffer, conventional ORBs lock the global freestore to prevent multiple threads from corrupting its internal freelist. However, the time required to allocate a new buffer depends on freestore fragmentation and memory management algorithms [19, 20]. Unless the freestore provides timeliness guarantees for buffer allocation and deletion, serializing access to the freestore can cause a higher priority thread to wait indefinitely, leading to unbounded priority inversion. As with connections, if priority inheritance is not supported by the mutual exclusion mechanism that locks the freestore, priority inversion can be further exacerbated.

One approach to alleviate this problem would be to use runtime stack allocation or a memory pool in thread-specific storage, which does not need mutual exclusion since neither is shared among multiple threads. In some use cases, however, it may not be possible to use these freestore implementations, *e.g.*, buffers need be shared by multiple threads in the ORB or the application.

Leader/Followers. Assume that the leader thread is of low priority, while the follower thread is of high priority. In steps 18 and 19, the leader thread handles the reply for the follower thread. During this time, the leader thread may be preempted by some other thread of medium priority before the reply is handed off to the follower. This priority inversion can be avoided if the leader thread can inherit the priority of the follower thread through a condition variable. Unfortunately, although most real-time operating systems support priority inheritance for mutexes, they rarely support priority inheritance for condition variables.

Reactor. There is no way to distinguish a high priority client request from one of lower priority at the reactor level in step 12. This lack of information can lead to unbounded priority inversion if a lower priority request is serviced before one of higher priority.

POA. In step 15, the POA dispatches the upcall after locating the target POA, servant, and skeleton. The time required to demultiplex the request may depend on the organization of the POA hierarchy and number of POAs, servants, or operations configured in the server. The time required to dispatch the request may depend on thread contention on a POA's dispatching table. Demultiplexing and dispatching in conventional CORBA implementations is typically inefficient and unpredictable [21, 5].

In summary, this section described the typical sources of unbounded priority inversion in the critical path of an ORB. Even though the details may vary between different ORB implementations, *e.g.*, an ORB implementation may use the Half-Sync/Half-Async concurrency pattern [16] instead of the Leader/Followers pattern, the key components of the ORB (*i.e.*, caching, memory management, concurrency, request demultiplexing, and dispatching components) must be evaluated carefully to identify potential sources of unbounded priority inversion.

2.3 Eliminating Sources of Unbounded Priority Inversion in TAO

To address the problems described in Section 2.2, we designed TAO's Real-time CORBA implementation to use non-multiplexed resources. In our design shown in Figure 3, each thread lane (described in Sidebar 2) has its own connection cache, memory pool freestore, acceptor, and leader/followers manager (which includes a reactor). These lane-specific resources are shared only by the threads in the lane. Priority

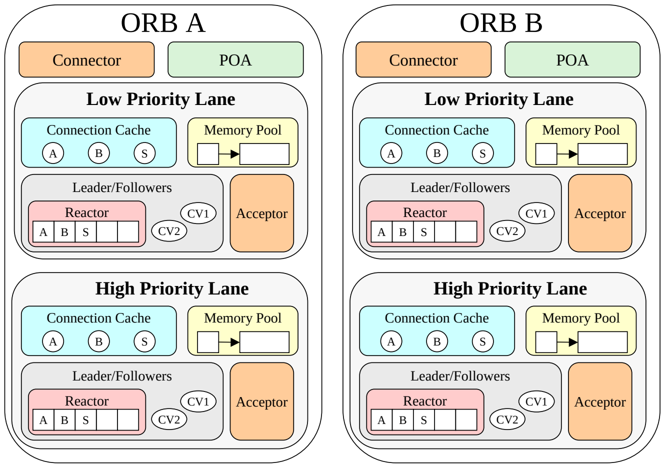


Figure 3: **The Design of the TAO Real-time CORBA**

inversion is therefore avoided since all the threads in a lane have the same priority.

All lanes in the TAO ORB share components that are not in the critical path (such as configuration factories) or that do not contribute to priority inversion (such as connectors). In standard CORBA, POAs are shared, which could be a source of non-determinism. However, our previous work on CORBA request demultiplexing [22] describes techniques to ensure predictable POA demultiplexing and reduce average- and worst-case overhead *regardless* of organization of an ORB's POA hierarchy or number of POAs, servants, or operations. Likewise, our work on efficient, scalable, and, predictable dispatching components [23] shows how it is possible to bound priority inversion and hence provide timeliness guarantees to DRE applications. Moreover, our work on Real-time CORBA explicit binding mechanisms illustrates how to design predictable connection management into an ORB [9].

One potential drawback with TAO's lane-specific resource technique is that it is more resource intensive than multiplexed techniques. For example, if a thread in a high priority lane has established a connection to a particular server, and a thread in a low priority lane wants to communicate with the same server, the low priority thread cannot use the connection cached in the high priority lane. DRE applications typically deal with a small range of distinct priorities, so TAO's lane-specific resource scheme is generally not a problem in practice. Moreover, when hundreds of priorities are desired, priorities can be banded into ranges to ease resource requirements and schedulability analysis of the system.

Sidebar 2: Real-time CORBA Thread Pools

Many real-time systems use multithreading to (a) distinguish between different types of service, such as high-priority vs. low-priority tasks [14], (b) support thread preemption to prevent unbounded priority inversion and deadlock, and (c) support complex object implementations that run for long and/or variable durations. To allow real-time ORB endsystems and applications to leverage these benefits of multithreading—while controlling the amount of memory and processor resources they consume—Real-time CORBA defines two types of server *thread pool* models [24]:

- *Thread pool with lanes* – In this model, threads in a pool are divided into *lanes* that are assigned different priorities. The priority of the threads generally does not change.
- *Thread pool without lanes* – In this model, all threads in a pool have the same assigned priority. However, the priority is changed to match the priority of a client making the request.

Each thread pool is then associated with one or more object adapters. The threads in a pool process client requests targeted at servants registered with the pool's associated object adapter(s).

2.4 Achieving End-to-end Predictability in TAO

After the ORB has been designed to avoid unbounded priority inversions in the critical path, end-to-end predictability can be achieved by propagating and preserving priorities. TAO accomplishes this by implementing the following Real-time CORBA features:

- **Thread pools**, which are described in Sidebar 2 and which are implemented in TAO [25]. TAO thread pools use the Leader/Followers design pattern [16] to reduce synchronization overhead and optimize memory management, thereby resulting in reduced priority inversions and improved dispatch latencies.
- **Priority propagation models**, which are described in Sidebar 3. The client priority propagation model allows a client to piggyback its priority with the request data to a server. A server thread processing this request changes its priority to match the priority of the client for the duration of its request processing. The server declared priority model allows the server to specify the priority at which requests on an object will be processed. This priority is specified when the object is registered with the object adapter and is stored in the object adapter's active object table. When a server thread processes a request on this object, its priority is changed to match the server declared priority for the duration of processing the request.
- **Explicit binding mechanisms**, which are described in Sidebar 4. Explicit binding enables clients to bind themselves to server objects using pre-allocated connections, priority bands, and private connections. TAO offers an implementation [11] of this feature for Real-time CORBA. TAO's connection cache is extended to keep track of the QoS properties of the connections, which includes priority banding and privacy attributes.

3 Experimentation Setup

This section describes the Real-time CORBA testbed and experiments used to evaluate TAO's end-to-end performance. We evaluate two different configurations of TAO:

- **Classic CORBA** — We first illustrate the limitations of classic CORBA implementations that cannot satisfy QoS requirements. This analysis is performed using a version of TAO configured without Real-time CORBA support.
- **Real-time CORBA** — We then perform the same experiments using Real-time CORBA features to show how end-to-end predictability can be achieved when the underlying middleware respects and propagates thread priorities, avoids unbounded priority inversions, and allows

Sidebar 3: Propagating Priorities with Real-time CORBA

Classic CORBA ORBs provide no standard mechanism for clients to indicate the relative priorities of their requests to ORB endsystems. This feature is necessary, however, to reduce end-to-end priority inversion, as well as to bound latency and jitter for applications with deterministic real-time QoS requirements. Real-time CORBA defines a *PriorityModel* policy with two values, SERVER_DECLARED and CLIENT_PROPAGATED, as described below:

- *Server declared priorities*—This model allows a server to dictate the priority at which an invocation on a particular object will execute. In the server declared model, the priority is designated *a priori* by the server and is encoded into the object reference published to the client.
- *Client propagated priorities*—This model allows a server to process incoming requests at a priority of the client thread. In this model, each invocation carries the CORBA priority of the operation in the service context list that is tunneled with its GIOP request. Each ORB endsystem along the activity path between the client and server processes the request at this priority. Moreover, if the client invokes a two-way operation, its CORBA priority will determine the priority of the reply.

Sidebar 4: Explicit Binding with Real-time CORBA

Before Real-time CORBA was standardized, the CORBA specification supported only *implicit binding*, where resources on the path between a client and its server object are established on-demand, *e.g.*, after a client's first invocation on the server. Implicit binding helps conserve OS and networking resources by deferring the creation of connections until they are actually used. Although not strictly part of the implicit binding model, many ORBs also allow multiple client threads in a process to be multiplexed through a shared network connection to their corresponding server.

Unfortunately, implicit binding and connection multiplexing are inadequate for DRE applications with stringent QoS requirements. In particular, deferring connection establishment until run-time can increase latency and jitter significantly. Moreover, the use of connection multiplexing can yield substantial priority inversion due to head-of-line blocking associated with connection queues that are processed in FIFO order.

To avoid these problems, the Real-time CORBA specification defines a group of orthogonal explicit binding mechanisms that support the following capabilities:

- A *connection pre-establishment* mechanism, which eliminates a common source of operation jitter
- A *PriorityBandedConnection* policy, where the invocation priority of the client determines which connection is used and
- A *PrivateConnection* policy, which guarantees non-multiplexed connections.

applications to configure and control processor, communication, and memory resources. This analysis is performed using a version of TAO that supports Real-time CORBA features.

By keeping the basic ORB capabilities (*e.g.*, stubs, skeletons, POA, reactor, acceptor, and connector) consistent, we can compare the impacts of eliminating unbounded priority inversions in the ORB and propagating priorities end-to-end.

3.1 Overview of the ORB Endsystem Testbed

Below we describe our ORB endsystem testbed as illustrated in Figure 4 and described in Table 1. Here we examine the client and server setup in terms of distribution, threading, scheduling, and priorities. We also describe how we vary load in our ORB endsystem testbed.

Client and server configuration. Most experiments collocate the client and server on the same machine and use a single CPU, as shown in Figure 4 (a). We focus our experiments on a single CPU hardware configuration in order to:

- Factor out differences in network-interface driver support and
- Isolate the impact of OS design and implementation on ORB middleware and application performance.

We explicitly mention when the client and server are distributed across different machines or when a second CPU is enabled, as shown in Figure 4 (b). When distributed, the client and server are connected by a 100 Mbps Ethernet.

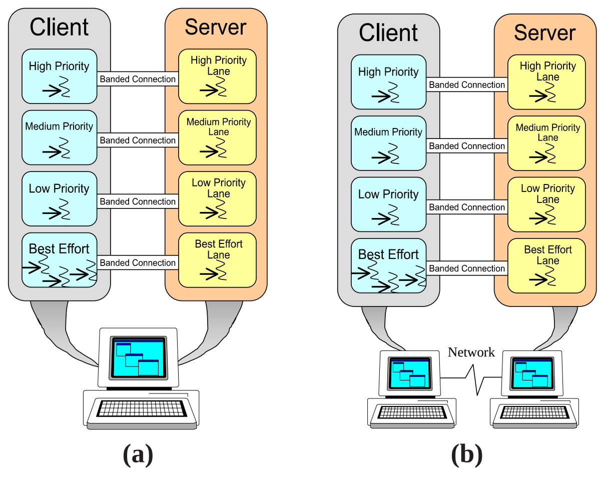


Figure 4: **Overview of Real-time CORBA Testbed**

As discussed in Section 3.2, clients in these experiments use two types of threads:

- Real-time threads of various priorities making invocations at periodic intervals and

Name	hermes.doc.wustl.edu
Hardware Profile	
OS	Linux 2.4 (Redhat 7.1)
Processor (2)	Intel Pentium III 930 MHz
Memory	500 Megabytes
CPU Cache	256 KB
Threads Profile	
ORBSchedPolicy	SCHED_FIFO
ORBScopePolicy	SYSTEM
ORBPriorityMapping	linear
Priority Profile	
High Priority Lane	32767
Medium Priority Lane	21844
Low Priority Lane	10922
Best Effort Lane	0

Table 1: **Description of Real-time CORBA Testbed**

- Best-effort threads that try to disrupt system predictability by stealing resources from real-time threads.

The servers in these experiments are configured with thread pools both with and without lanes.

Rate-based threads. Rate-based threads are identified by their frequency of invocation. For example, an H Hertz thread tries to make H invocations to the server every second. The period P of a rate-based thread is the multiplicative inverse of its frequency. E is the time it takes for an invocation to complete and it depends on the `work` and the QoS it receives from the client ORB endsystem, the network, and the server ORB endsystem.

The following are three scenarios of an invocation's execution with respect to its period:

- **Invocation completes within period.** In this scenario, a rate-based thread sleeps for time S equal to $(P - E)$ before making its next invocation, as shown in Figure 5. No deadlines are missed in this case.

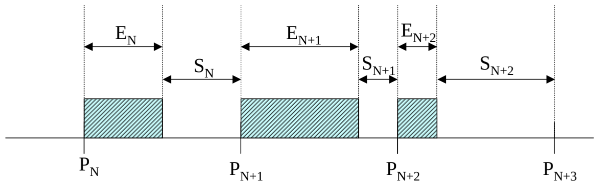


Figure 5: **Invocation Completes Within Its Period**

- **Invocation execution time exceeds period.** In this scenario, the execution time of invocation N (E_N) is such that invocation $N+1$ is invoked immediately since there is no time to sleep, *i.e.*, $(P - E) \leq 0$, as shown in Figure 6. Note that since the invocations are synchronous two-way CORBA calls, in any given thread a second invocation can only be made after the first one completes. In presenting the performance results, invocations N and

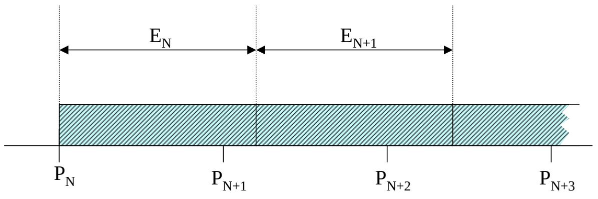


Figure 6: **Invocation Takes Longer Than Its Period**

$N + 1$ are not considered to have missed their deadlines. An invocation is considered to have missed its deadline only when it cannot be started within its period. [4] If the execution time is consistently greater than the period, however, then $\frac{E-P}{E}$ fraction of the deadlines will be missed.

- **Invocation misses deadline.** In this scenario, the execution time of invocation N (E_N) is such that invocation $N + 1$ could not be made during time P_{N+1} through P_{N+2}, as shown in Figure 7. In this case, invocation

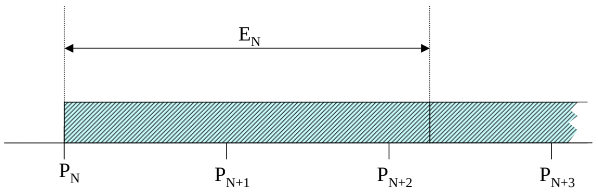


Figure 7: **Invocation Misses Deadline**

$N + 1$ missed its deadline and was not invoked.

Continuous threads. Continuous client threads use a flooding model of invocations on a server, *i.e.*, they do not pause between successive invocations.

FIFO scheduling. Threads used in the real-time experiments are placed in the OS FIFO scheduling class and system scheduling scope. The FIFO scheduling class provides the most predictable behavior since threads scheduled using this policy will run to completion unless they are preempted by a higher priority thread.

Linear mapping. CORBA priorities are linearly mapped to native OS priorities and vice versa. For example, on Linux 2.4[5] the `RTCORBA::maxPriority` of 32,767 maps to the maximum priority in the FIFO scheduling class of 99, `RTCORBA::minPriority` of 0 maps to the minimum priority in the FIFO scheduling class of 1, and everything in between is evenly apportioned. The CORBA priority range is

[4]These experiments can also be configured with more stringent requirements, *e.g.*, where an invocation is considered to miss its deadline when it cannot complete execution within its period.

[5]Equivalent performance results were obtained when these experiments were conducted on Solaris 2.7 and similar behavior is expected on other platforms with comparable real-time characteristics.

divided evenly such that the priority lanes are assigned the following CORBA priorities:

- The high priority thread lane is assigned 32,767;
- The medium priority lane is assigned 21,844;
- The low priority lane is assigned 10,922; and
- The best-effort thread lane is assigned 0.

This testbed utilizes four different thread priorities, which is representative of the fixed-priority hard real-time systems (such as avionics mission computing [26]) for which Real-time CORBA 1.0 [2] was designed. Our future work on Real-time CORBA 2.0 [27] is exploring dynamic scheduling capabilities [28] that handle a broader range of priorities.

Operation invocation. The signature of the operation invoked by the client on the server is:

```
void method (in unsigned long work);
```

The `work` parameter specifies the amount of CPU intensive work the server will perform to service this invocation. The higher the value of `work`, therefore, the more the number of iterations of a CPU consuming loop in the servant upcall, resulting in increased load on the server.

3.2 Overview of the Real-time ORB Experiments

The experiments presented here measure the degree of real-time, deterministic, and predictable behavior of CORBA middleware. End-to-end predictability of timeliness in a fixed priority CORBA system is defined as:

- Respecting thread priorities for resolving resource contention during the processing of CORBA invocations.
- Bounding the duration of thread priority inversions during end-to-end processing.
- Bounding operation invocations latencies.

The following experiments demonstrate the degree of end-to-end predictability exhibited by different ORB configurations by showing the extent to which they can propagate and preserve priorities, exercise control over the management of resources, and avoid unbounded priority inversions. Our experiments vary different aspects and parameters of the ORB endsystem testbed (described in Section 3.1) to measure how well these conditions are met by the ORB middleware. The following nine experiments measure system reaction to increased workload, ranging from unloaded to an overloaded situation, as well as ORB endsystem reaction to increased best-effort work:

1. Increasing workload
2. Increasing invocation rate
3. Increasing client and server concurrency
4. Increasing workload in classic CORBA
5. Increasing workload in Real-time CORBA with lanes
   Increasing priority $\rightarrow$ Increasing rate
6. Increasing workload in Real-time CORBA with lanes
   Increasing priority $\rightarrow$ Decreasing rate
7. Increasing best-effort work in classic CORBA
8. Increasing best-effort work in Real-time CORBA with lanes
9. Increasing workload in Real-time CORBA without lanes

The first three experiments are basic non-real-time tests that provide a baseline for general characteristics of the ORB endsystem testbed. All threads in these three experiments have the default OS priority and are scheduled in the default scheduling class. Experiment 1 measures throughput as the workload increases. Experiment 2 measures deadlines made/missed as the target invocation rate goes beyond the ORB endsystem capacity. Experiment 3 measures throughput as client and server concurrency increases, while also adding additional CPU support.

The remaining six experiments contain several client threads of different importance. The importance of each thread is mapped to its relative priority. Experiments 4–6 measure throughput with and without Real-time CORBA as the workload increases. Experiments 7 and 8 measure throughput with and without Real-time CORBA, as best-effort work increases. Finally, Experiment 9 measures throughput with Real-time CORBA thread pools without lanes as workload increases. Experiments 5, 6, and 8 use Real-time CORBA thread pools with lanes.

All the Real-time CORBA experiments exercise the CLIENT_PROPAGATED policy to preserve end-to-end priority. The priority of threads in a thread pool with lanes is fixed. However, the priority of threads in a thread pool without lanes is adjusted to match the priority of the client when processing the CORBA request. After processing completes, the ORB restores the thread's priority to the priority of the thread pool.

4 Experimentation Results

This section presents the results of an empirical analysis of end-to-end ORB behavior based on the testbed described in Section 3. We describe the configurations used and the performance results obtained for the various real-time CORBA experiments we conducted.

Experiment 1: Increasing Workload

This experiment measures the effect of increasing workload in the ORB endsystem testbed. As shown in Figure 8, the server has one thread handling incoming requests and the client has

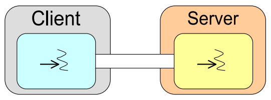


Figure 8: **Configuration for Measuring the Effect of Increasing Workload**

one continuous thread making invocations. Workload is increased by increasing the `work` parameter in the invocation, which makes the server perform more work for every client request. The performance graph in Figure 9 plots the throughput achieved (invocations/second) as the workload increases. Not surprisingly, as the workload increases, the throughput decreases.

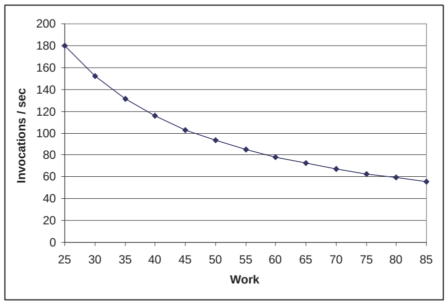


Figure 9: **Effect of Increasing Workload**

Experiment 2: Increasing Invocation Rate

This experiment measures the impact of increasing the target frequency for a rate-based thread beyond the capacity of the system. As shown in Figure 10, the server has one thread handling incoming requests and the client has one rate-based thread making invocations. As the frequency of the rate-based

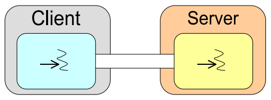


Figure 10: **Configuration for Measuring the Effect of Increasing Invocation Rate**

thread is increased, it eventually exceeds the capacity of the system. Workload is held constant in this experiment at 30. Figure 9 shows that in Experiment 1, the continuous thread could make ∼150 invocations/second with a workload of 30. ORB endsystem capacity is therefore estimated at ∼150 invocations/second with this workload.

The performance graph in Figure 11 plots the percentage of deadlines made as the target frequency of the rate-based thread increases. Until the target frequency increases to ∼150 invocations/second, the rate-based thread is able to meet 100% of its deadlines. After the target frequency goes beyond 150 invocations/second, however, the rate-based thread starts to miss deadlines. The number of deadlines missed increases with the increased target frequency.

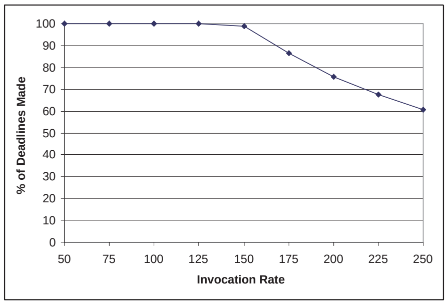


Figure 11: **Effect of Increasing Invocation Rate**

Experiment 3: Increasing Client and Server Concurrency

This experiment measures the impact of increasing the concurrency of both the client and the server as shown in Figure 12. The following three server configurations are used in this experiment:

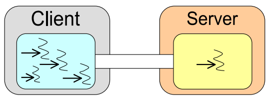


Figure 12: **Configuration for Measuring the Effect of Increasing Concurrency**

1. One thread to handle incoming requests; one CPU utilized.
2. Two threads to handle incoming requests; one CPU utilized.
3. Two threads to handle incoming requests; two CPUs utilized.

For each of these server configurations, the number of client threads making continuous invocations is increased from 1 to 20. As with Experiment 2, workload is held constant at 30.

The performance graph in Figure 13 plots the collective/cumulative throughput achieved for all the client threads as the number of client threads increases. The results are described below.

Configuration 1 (one server thread, one CPU). Increasing client concurrency does not impact collective throughput of the client threads. Thus, as the number of client threads increases, the throughput per thread decreases, but the collective throughput remains constant. This result occurs because the server performs most of the processing, so increasing client

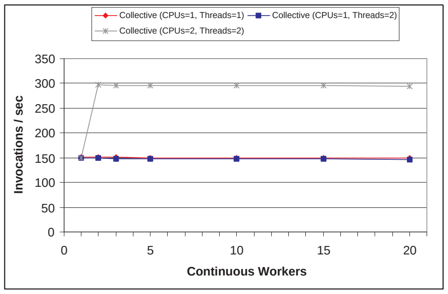


Figure 13: **Effect of Increasing Concurrency**

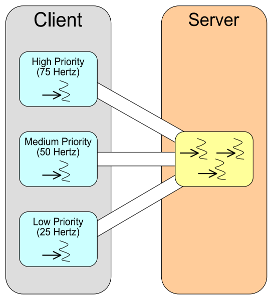


Figure 14: **Configuration for Measuring the Effect of Increasing Workload in Classic CORBA**

concurrency does not impact the collective throughput for the client threads.

Configuration 2 (two server threads, one CPU). The results are almost identical to server configuration 1 (one server thread, one CPU). Increasing server concurrency without improving hardware support does not improve throughput when the work is CPU bound. In fact, the throughput degrades slightly since the server must now coordinate and synchronize the two threads on one CPU.

Configuration 3 (two server threads, two CPUs). After the number of client threads reaches two, the collective throughput doubles since the second client thread engages the second server thread, thereby doubling the throughput.[6] Increasing the number of client threads further does not improve collective throughput since both server threads are already engaged.

Experiment 4: Increasing Workload in Classic CORBA

This experiment measures the disruption caused by increasing workload in classic CORBA. As shown in Figure 14, the server has three threads handling incoming requests.[7] The client has three rate-based threads running at different importance levels, *i.e.*, the high priority thread runs at 75 Hertz, the medium priority thread runs at 50 Hertz, and the low priority thread runs at 25 Hertz.

The performance graph in Figure 15 plots the throughput achieved for each of the three client threads as the workload increases. The combined capacity desired by the three client threads is 150 invocations/second (75 + 50 + 25). Correlating this desired throughput of 150 invocations/second to the performance graph in Experiment 1 (Figure 9), note that the continuous thread achieved that throughput with a workload of 30 or less. Therefore, each of the three client threads achieved their desired frequency for workloads of 25 and 30 in Figure 15 since the ORB endsystem capacity was not exceeded. After the workload increased beyond 30, however, deadlines

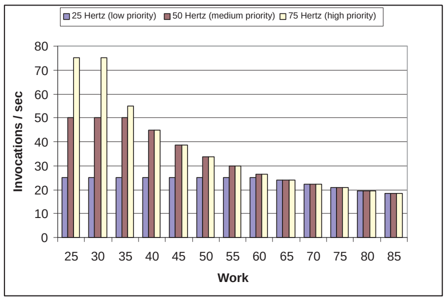


Figure 15: **Effect of Increasing Workload in Classic CORBA**

start being missed. The expected behavior of most DRE applications is to drop requests from client threads of lower priority before dropping requests from those of higher priority. Unfortunately, since all three clients are treated equally by the classic CORBA server, the first to be affected is the high priority 75 Hertz client thread, followed by the medium priority 50 Hertz client thread, and finally by the low priority 25 Hertz client thread. This behavior is clearly unacceptable for most DRE applications.

Experiment 5: Increasing Workload in Real-time CORBA with Lanes: Increasing Priority $\rightarrow$ Increasing Rate

This experiment measures the disruption caused by increasing workload in Real-time CORBA with lanes. As shown in Figure 16, the server has three thread lanes of high, medium, and low priorities to handle incoming requests. Each lane has one thread. The client is identical to the client in Experiment 4.

The performance graph in Figure 17 plots the throughput achieved for each of the three client threads as the workload increases. Each of the three client threads achieved its desired

[6] Some applications may not be able to double the throughput using this configuration if synchronization is required in the servant code while processing the CORBA requests.

[7] It is not necessary to have three threads handling incoming requests on the server. As shown in Figure 13, one thread is sufficient since increasing server concurrency without increasing CPU support does not improve throughput when the work is CPU bound.

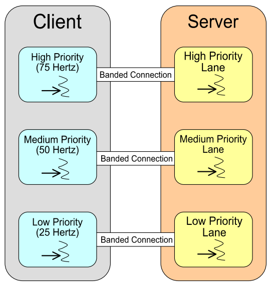
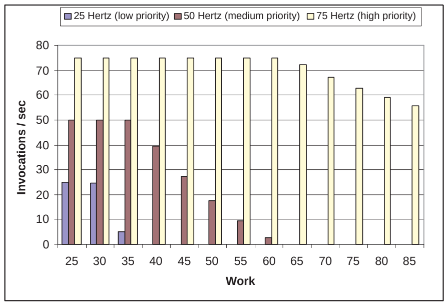


Figure 16: **Increasing Workload in Real-time CORBA (Increasing Priority → Increasing Rate): Configuration of Testbed**

frequency for workloads of 25 and 30 since the ORB endsystem capacity has not been exceeded. After the workload



,

Figure 17: **Effect of Increasing Workload in Real-time CORBA: Increasing Priority → Increasing Rate (Client and Server are on the Same Machine)**

increases beyond 30, however, deadlines start being missed. Unlike the classic CORBA experiment (Figure 15), the first deadline affected is the low priority 25 Hertz client thread, followed by the medium priority 50 Hertz client thread, and finally the high priority 75 Hertz client thread. This behavior is what a DRE application expects.

Figure 18 shows the performance graph of the same experiment, except the client and server are distributed on two different machines on the network. The results are similar to those obtained when the client and server are on the same machine (Figure 17). However, between the time the high priority server thread sent a reply to the high priority client thread and before it receives a new request from it, the server can process a request from a client thread of lower priority. The medium priority 50 Hertz client thread can therefore make some progress.

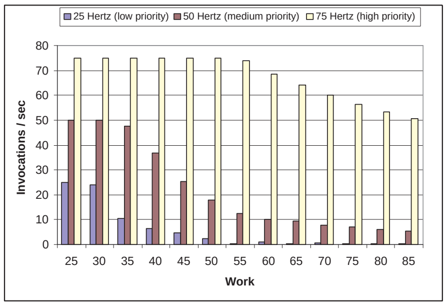


Figure 18: **Effect of Increasing Workload in Real-time CORBA: Increasing Priority → Increasing Rate (Client and Server are Remote)**

Experiment 6: Increasing Workload in Real-time CORBA with Lanes: Increasing Priority → Decreasing Rate

This experiment is similar to Experiment 5 except that the high priority thread runs at 25 Hertz, the medium priority thread runs at 50 Hertz, and the low priority thread runs at 75 Hertz, as shown in Figure 19. The performance graph in Figure 20

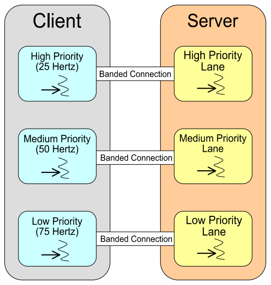


Figure 19: **Configuration for Measuring the Effect of Increasing Workload in Real-time CORBA: Increasing Priority → Decreasing Rate**

shows that the low priority 75 Hertz client thread is affected first, followed by the medium priority 50 Hertz client thread. The high priority 25 Hertz client thread is unaffected since the ORB endsystem capacity never dropped below 25 invocations/second. This behavior is expected from a DRE application.

Experiment 7: Increasing Best-effort Work in Classic CORBA

This experiment measures the disruption caused by increasing best-effort work in classic CORBA. As shown in Figure 21, the server has four threads handling incoming requests.[8] The client has three rate-based threads of different priorities, *i.e.*,

[8]For the same reasons noted in Experiment 4, one server thread is sufficient for handling incoming requests.

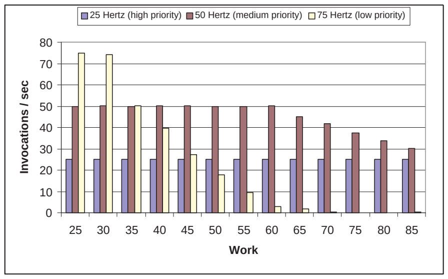


Figure 20: **Effect of Increasing Workload in Real-time CORBA: Increasing Priority → decreasing rate**

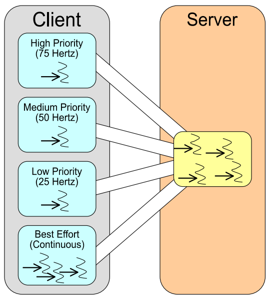


Figure 21: **Configuration for Measuring the Effect of Increasing Best-effort Work in Classic CORBA**

the high priority thread runs at 75 Hertz, the medium priority thread runs at 50 Hertz, and the low priority thread runs at 25 Hertz. The client also has a variable number of best-effort threads making continuous invocations. Workload is held constant at 30 and the number of best-effort continuous client threads is increased from 0 through 10. Recall from Experiment 1 that the ORB endsystem capacity is 150 invocation/second for a workload of 30. Any progress made by the best-effort continuous client threads will therefore cause the rate-based threads to miss their deadlines.

The performance graph in Figure 22 plots the throughput achieved for each of the three rate-based client threads and the collective throughput achieved by the all the best-effort continuous threads on the client as the number of best-effort continuous threads increases. When there are no best-effort threads, the three rate-based threads achieve their desired frequency. After best-effort threads are added to the client, however, the classic CORBA server treats all the client threads equally, so the collective throughput of the best-effort threads increases at the expense of the higher priority threads. This behavior is clearly unacceptable for most DRE applications.

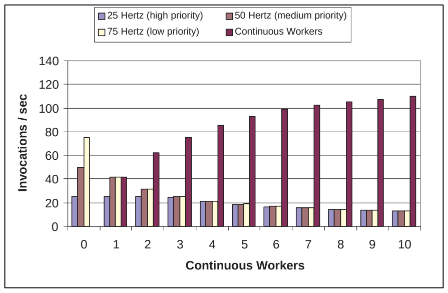


Figure 22: **Effect of Increasing Best-effort Work in Classic CORBA**

Experiment 8: Increasing Best-effort Work in Real-time CORBA with Lanes

This experiment measures the disruption caused by increasing best-effort work in Real-time CORBA configured to use thread pools with lanes. As shown in Figure 23, the server processes requests using four thread lanes of high, medium, low, and best-effort priorities. Each lane has one thread. The client is identical to the client in Experiment 7.

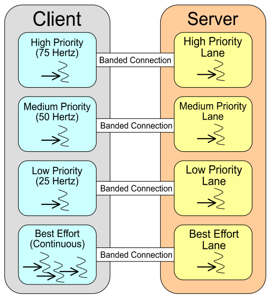


Figure 23: **Configuration for Measuring the Effect of Increasing Best-effort Work in Real-time CORBA**

The performance graph in Figure 24 shows that best-effort continuous threads do not affect the higher priority rate-based threads. This behavior is expected from a DRE application.

Figure 25 shows the performance graph of the same experiment with a slightly lower workload `(work = 28)`. Note that the slack produced by the lower workload is used by the best-effort threads. Moreover, increasing the number of best-effort threads does not yield any increase in the collective throughput of the best-effort threads and the best-effort threads are not able to disrupt the higher priority rate-based threads. This behavior is expected from a DRE application.

Figure 26 shows the performance graph of the same experiment `(work = 28)`, but with the client and server on different machines across a network. Note that the slack available for the best-effort threads increases since all the client processing is now performed on the machine hosting the client,

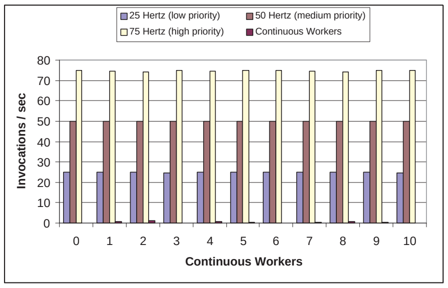


Figure 24: **Effect of Increasing Best-effort Work in Real-time CORBA: System Running at Capacity (Work = 30); Client and Server are on the Same Machine**

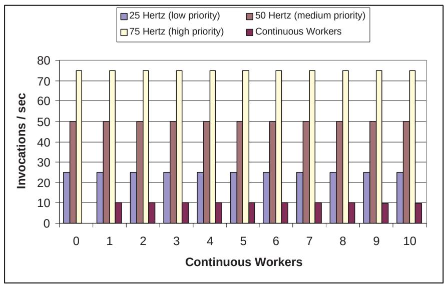


Figure 25: **Effect of Increasing Best-effort Work in Real-time CORBA: System Running Slightly Below Capacity (Work = 28); Client and Server are on the Same Machine**

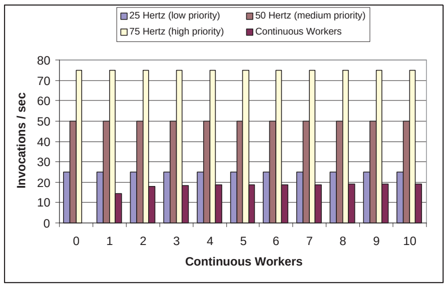


Figure 26: **Effect of Increasing Best-effort Work in Real-time CORBA: System Running Slightly Below Capacity (Work = 28); Client and Server are Remote**

freeing up the server to do additional work.

Experiment 9: Increasing Workload in Real-time CORBA without Lanes

Our final experiment measures the disruption caused by increasing workload in Real-time CORBA without lanes. As shown in Figure 27, the server has a thread pool of three threads handling incoming requests. The client is identical to the client in Experiments 4 and 5.

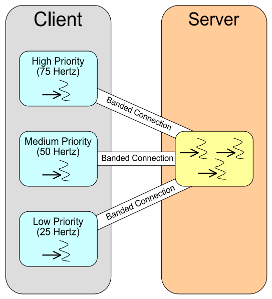


Figure 27: **Configuration for Measuring the Effect of Increasing Workload in Real-time CORBA without Lanes**

The performance graphs in Figures 28, 29, and 30 plot the throughput achieved for each of the three client threads as the workload increases. The difference between the three graphs is the priority of the server thread pool. In Figure 28 the thread pool runs at low priority, in Figure 29 the thread pool priority runs at medium priority, and in Figure 30 the thread pool runs at high priority. Below, we discuss each of these results in more depth.

Server thread pool priority = low. Assume that one of the server threads is processing a request from a client thread of low priority. During that time, a request arrives at the server from a higher priority client thread. Unfortunately, since the request-processing thread has the same priority as the waiting thread, the waiting thread is unable to preempt the processing thread. The request from the higher priority client thread must therefore wait until the low priority client thread request has been processed completely, which leads to the three client threads being treated equally by the server, as shown in Figure 28. This behavior is clearly unacceptable for most DRE applications.

Server thread pool priority = medium. Assume that one of the server threads is processing a request from a client thread of medium priority. During this time, a request arrives at the server from the high priority client thread. Unfortunately, since the request-processing thread has the same priority as the waiting thread, the waiting thread is unable to preempt the request processing thread. The request from the high priority client thread must therefore wait until the medium priority

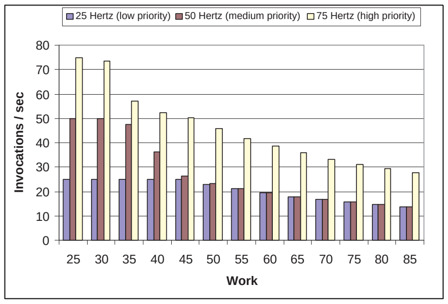


Figure 28: **Effect of Increasing Workload in Real-time CORBA Without Lanes: Server Thread Pool Priority = Low**

client thread request has been processed completely. When a medium or high priority request arrives while a low priority request is being processed, however, the behavior is different. Since the priority of the waiting thread is greater than that of the priority of the request processing thread, it can preempt the request processing thread and handle the higher priority request.

This behavior leads to the medium and high priority client threads being treated equally but are given preference over the low priority client thread, as shown in Figure 29. This behavior is also unacceptable for a DRE application.

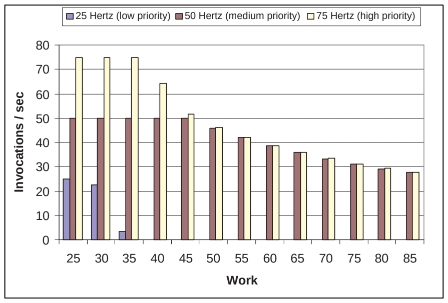


Figure 29: **Effect of Increasing Workload in Real-time CORBA without Lanes: Server Thread Pool Priority = Medium**

Server thread pool priority = high. Assume that one of the server threads is processing a request from a client thread of low priority. During this time, a request arrives at the server from a higher priority client thread. Since the priority of the waiting thread is greater than the priority of the request processing thread, it can preempt the request-processing thread and handle the higher priority request. Note that a high priority request can preempt both low and medium priority requests.

This behavior ensures (1) the high priority client thread is preferred over the low and medium priority client threads and (2) the medium priority client thread is preferred over the low priority client thread, as shown in Figure 30. This behavior is expected from a DRE application.

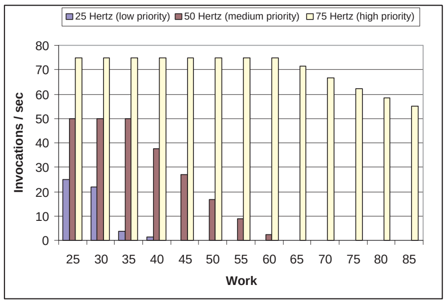


Figure 30: **Effect of Increasing Workload in Real-time CORBA without Lanes: Server Thread Pool Priority = High**

Notes on using Real-time CORBA without lanes. Experiment 9 showed that the most desirable behavior is achieved in Figure 30 where the server thread pool priority is equal to the highest request processing priority. However, a server using a thread pool without lanes can incur priority inversion. Consider the case where the server thread pool priority is high. Assume that one of the server threads is processing a request from a client thread of medium priority. During this time, a request arrives at the server from the low priority client thread. Since the priority of the waiting thread is greater than the priority of the request processing thread, it is able to preempt the request processing thread to read the incoming request. If the request includes a large amount of data, it can take a significant amount of time to read the request. After this thread reads the request, it sets its priority low to match the client propagated priority. The medium priority request-processing thread is now able to preempt the low priority thread and resume processing.

The interruption caused by the low priority request leads to priority inversion for the medium priority thread. Depending on the number of waiting threads in the server thread pool and the time it takes to read the incoming request, the priority inversion can be significant or even unbounded.

4.1 Summary of Empirical Results

Experiments 4 and 7 illustrate that classic CORBA implementations cannot preserve priorities end-to-end, which makes them inappropriate for DRE applications with strict predictability requirements. Conversely, the experiments that use TAO with Real-time CORBA features enabled (Experiments 5, 6, 8, and 9) preserve priorities end-to-end since TAO was able to

- Avoid unbounded priority inversions in the critical path, as discussed in Section 2.3 and

- Propagate and preserve priorities end-to-end, as discussed in Section 2.4.

Thread pools in Real-time CORBA are an important feature that allows application developers and end-users to configure and control processor resources. A server using thread pools without lanes is more flexible than a server using thread pools with lanes since the former can adapt to any priority propagated by the client. As we have shown, however, there are cases when this configuration can incur priority inversion. A client invoking requests on a server using thread pools with lanes is restricted to priorities used in the lanes by the server.[9] This configuration provides the most predictable execution and does not exhibit unbounded priority inversions.

5 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 distribution middleware, such as CORBA. In this section, we compare our work on Real-time CORBA and TAO with related work.

Dynamic Scheduling Services and Resource Managers. Real-time CORBA is well suited for applications using fixed priority scheduling. However, for applications that execute under dynamic load conditions [28] and cannot determine the priorities of various operations *a priori* without significantly underutilizing various resources, the Object Management Group (OMG) is standardizing dynamic scheduling [29] techniques, such as deadline-based [30] or value-based [31] scheduling. Work is currently underway to integrate TAO with dynamic scheduling services, such as the Kokyu scheduling framework [32], and adaptive resource management systems, such as the RT-ARM [33] framework.

Timed Distributed Invocations. 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) [34]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed method invocations (`TDMI`s) [35]. A difference between TAO and the URI approaches is that `TDMI`s express required timing constraints, *e.g.,* deadlines relative to the current time, whereas the TAO ORB is based on the fixed-priority scheduling features defined in the Real-time CORBA specification.

ROFES. ROFES [36] is a Real-Time CORBA implementation for embedded systems with a microkernel like architecture. ROFES has been adapted to work with several different hard real-time networks, including SCI [37], CAN, ATM, and a ethernet-based time-triggered protocol [38].

Similar work has been done in the context of TAO for developing a real-time I/O (RIO) [39] subsystem optimized for QoS-enabled endsystems that support high-performance and real-time applications running on off-the-shelf hardware and software.

Quality Objects. The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [40]. QuO is based on CORBA and provides the following support for QoS-enabled applications:

- *Run-time performance tuning and configuration* through the specification of operating regions, behavior alternatives, and reconfiguration strategies that allows the QuO run-time to trigger reconfiguration adaptively as system conditions change (represented by transitions between operating regions); and
- *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.

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) [41]. The QuO middleware adds significant value to adaptive real-time ORBs such as TAO. We are currently collaborating [42] with the BBN QuO team to integrate the TAO and QuO middleware as part of the DARPA Quorum project [43].

6 Concluding Remarks

The Real-time CORBA 1.0 specification [2] has introduced several novel concepts and requirements to the CORBA object model. These new requirements for providing end-to-end predictability have added to the challenges faced by the ORB developers. This paper described how we identified and eliminated sources of unbounded priority inversion in the critical code path of the TAO ORB. TAO's Real-time CORBA implementation uses non-multiplexed resources where possible. Moreover, it is designed to bound priority inversions for resources that must be shared.

The empirical analysis presented in this paper illustrated that ORBs lacking Real-time CORBA features are unsuitable for DRE applications that require predictable behavior. The experimental results also validate the end-to-end real-time behavior of TAO. Even when using a real-time ORB, however, careful consideration must be given to several other factors

[9] Priority bands can be used to map a range of client priorities to a server lane. However, the server lane priority is fixed and therefore, request processing priorities are limited to the priorities of the lanes.

to ensure end-to-end system predictability and scalability, including:

- The configuration and structure of the client and server
- The use of lanes in thread pools
- The priorities assigned to the thread pools and lanes used in the server and
- The priority propagation and banding policies used.

In general, thread pools with lanes are not as flexible as their counterparts without lanes because the priority of the threads in lanes are fixed. However, thread pools with lanes provide the most predictable execution and do not exhibit unbounded priority inversions.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.6*, Dec. 2001.

[2] Object Management Group, *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., Mar. 1999.

[3] 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*, vol. 33, June 2000.

[4] C. O'Ryan, D. C. Schmidt, and J. R. Noseworthy, "Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations," *International Journal of Computer Systems Science and Engineering*, vol. 17, Mar. 2002.

[5] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[6] 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.

[7] 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.

[8] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3^{rd} Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[9] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. L. Levine, "Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA," *Concurrency and Computing: Practice and Experience*, vol. 13, no. 2, pp. 507–541, 2001.

[10] 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.

[11] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0," in *Proceedings of the 6^{th} IEEE Real-Time Technology and Applications Symposium*, (Washington DC), IEEE, May 2000.

[12] 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.

[13] Center for Distributed Object Computing, "The ACE ORB (TAO)." www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[14] 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), pp. 184–199, ACM, Oct. 1997.

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

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

[17] I. Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter," *ACM StandardView*, vol. 6, Mar. 1998.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-time Synchronization," *IEEE Transactions on Computers*, vol. 39, Sept. 1990.

[19] S. M. Donahue, M. P. Hampton, M. Deters, J. M. Nye, R. K. Cytron, and K. M. Kavi, "Storage allocation for real-time, embedded systems," in *Embedded Software: Proceedings of the First International Workshop* (T. A. Henzinger and C. M. Kirsch, eds.), pp. 131–147, Springer Verlag, 2001.

[20] S. Donahue, M. Hampton, R. Cytron, M. Franklin, and K. Kavi, "Hardware support for fast and bounded-time storage allocation," *Second Annual Workshop on Memory Performance Issues (WMPI 2002)*, 2002.

[21] 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, Aug. 1996.

[22] 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.

[23] I. Pyarali, C. O'Ryan, and D. C. Schmidt, "A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2000.

[24] 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*, vol. 21, no. 2, 2001.

[25] I. Pyarali, M. Spivak, R. K. Cytron, and D. C. Schmidt, "Optimizing Threadpool Strategies for Real-Time CORBA," in *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, (Snowbird, Utah), pp. 214–222, ACM SIGPLAN, June 2001.

[26] 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.

[27] Object Management Group, *Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission*, OMG Document orbos/2001-06-09 ed., Apr. 2001.

[28] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, Mar. 2001.

[29] Object Management Group, *Dynamic Scheduling*, OMG Document orbos/99-03-32 ed., Mar. 1999.

[30] Y.-C. Wang and K.-J. Lin, "Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," in *IEEE Real-Time Systems Symposium*, pp. 246–255, IEEE, Dec. 1999.

[31] E. D. Jensen, "Eliminating the Hard/Soft Real-Time Dichotomy," *Embedded Systems Programming*, vol. 7, Oct. 1994.

[32] C. D. Gill, R. Cytron, and D. C. Schmidt, "Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems," in *Proceedings of the* 7^{th} *Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002.

[33] J. Huang and R. Jha and W. Heimerdinger and M. Muhammad and S. Lauzac and B. Kannikeswaran and K. Schwan and W. Zhao and R. Bettati, "RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications," in *Workshop on Middleware for Distributed Real-Time Systems, RTSS-97*, (San Francisco, California), IEEE, 1997.

[34] 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.

[35] 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.

[36] RWTH Aachen, "ROFES." http://www.rofes.de, 2002.

[37] M. P. S. Lankes and T. Bemmerl, "Design and Implementation of a SCI-based Real-Time CORBA," in *4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, (Magdeburg, Germany), IEEE, May 2001.

[38] M. R. S. Lankes and A. Jabs, "A Time-Triggered Ethernet Protocol for Real-Time CORBA," in *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, (Washington, DC), IEEE, Apr. 2002.

[39] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsystems," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, (Edinburgh, Scotland), OMG, Sept. 1999.

[40] 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, pp. 1–20, 1997.

[41] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[42] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pp. 625–634, IEEE, Apr. 2001.

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