Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers

Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale
{schmidt,sumedh,sergio,gokhale}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

Abstract

There is increasing demand to extend Object Request Broker (ORB) middleware to support applications with stringent real-time requirements. However, conventional ORBs, such as OMG CORBA, exhibit substantial priority inversion and non-determinism, which makes them unsuitable for applications with deterministic real-time requirements. This paper provides two contributions to the study and design of real-time ORB middleware. First, it illustrates empirically why conventional ORBs do not yet support real-time quality of service. Second, it describes software architectures that reduce priority inversion and non-determinism in real-time CORBA ORBs. The results presented in this paper demonstrate the feasibility of using standard OO middleware like CORBA over COTS hardware and software.

Keywords: Real-time CORBA Object Request Broker, QoS-enabled OO Middleware, Performance Measurements

1 Introduction

1.1 Emerging Trends in Distributed Real-time Systems

Next-generation distributed and real-time applications, such as video-on-demand, teleconferencing, and avionics, require endsystems that can provide statistical and deterministic quality of service (QoS) guarantees for latency [1], bandwidth, and reliability [2]. The following trends are shaping the evolution of software development techniques for these distributed real-time applications and endsystems:

Increased focus on middleware and integration frameworks: There is a general industry trend away from *programming* real-time applications from scratch to *integrating* applications using reusable components based on object-oriented (OO) middleware [3].

Increased focus on QoS-enabled components and open systems: There is increasing demand for remote method invocation and messaging technology to simplify the collaboration of open distributed application components [4] that possess stringent QoS requirements.

Increased focus on standardizing real-time middleware: Several international efforts are currently addressing QoS for OO middleware. The most prominent is the OMG CORBA standardization effort [5]. CORBA is OO middleware that allows clients to invoke operations on objects without concern for where the objects reside, what language the objects are written in, what OS/hardware platform they run on, or what communication protocols and networks are used to interconnect distributed objects [6].

There has been recent progress towards standardizing [7, 8] real-time CORBA. Several OMG groups, most notably the Real-Time Special Interest Group (RT SIG), are actively investigating standard extensions to CORBA to support distributed real-time applications. The intent of the real-time CORBA standardization effort is to enable real-time applications to interwork throughout embedded systems and heterogeneous distributed environments.

Notwithstanding the significant efforts of the OMG RT SIG, however, developing and standardizing distributed real-time CORBA ORBs remains hard. There are few successful exemplars of standard, commercially available distributed real-time ORB middleware. In particular, conventional CORBA ORBs are not well suited for performance-sensitive, distributed real-time applications due to (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) general lack of performance and predictability [9].

Although some operating systems, networks, and protocols now support real-time scheduling, they do not provide integrated end-to-end real-time ORB endsystem solutions [10]. Moreover, relatively little systems research has focused on

*This work was supported in part by Boeing, CDI, DARPA contract 9701516, Lucent, Motorola, NSF grant NCR-9628218, Siemens, and US Sprint.

strategies and tactics for real-time CORBA. In particular, QoS research at the network and OS layers has not addressed key requirements and programming aspects of CORBA middleware. For instance, research on QoS for ATM networks has focused largely on policies for allocating bandwidth on a virtual circuit basis [11]. Likewise, research on real-time operating systems has focused largely on avoiding priority inversions in synchronization and dispatching mechanisms for multi-threaded applications [12].

1.2 Towards Real-time CORBA

We believe that developing real-time OO middleware requires a systematic, measurement-driven methodology to identify and alleviate sources of ORB endsystem overhead, priority inversion, and non-determinism. The ORB software architectures presented in this paper are based on our experience developing, profiling, and optimizing next-generation avionics [13] and telecommunications [14] systems using real-time OO middleware such as ACE [15] and TAO [10].

ACE is an OO framework that implements core concurrency and distribution patterns [16] for communication software. It provides reusable C++ wrapper facades and framework components that support high-performance, real-time applications. ACE runs on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like VxWorks, Chorus Classix, pSoS, and LynxOS.

TAO is a highly extensible, ORB endsystem written using ACE. It is targeted for applications with deterministic and statistical QoS requirements, as well as best effort requirements. TAO is fully compliant with the latest OMG CORBA specifications [17] and is the first standard CORBA ORB endsystem that can support end-to-end QoS guarantees over ATM networks.

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

- Identifying enhancements to standard ORB specifications, such as OMG CORBA, that will enable applications to specify their QoS requirements concisely to ORB endsystems [18].
- Empirically determining the features required to build real-time ORB endsystems that can enforce deterministic and statistical end-to-end applications QoS guarantees [10].
- Integrating the strategies for I/O subsystem architectures and optimizations [19] with ORB endsystems to provide end-to-end bandwidth, latency, and reliability guarantees to distributed applications.
- Capturing and documenting the key design patterns [20] necessary to develop, maintain, configure, and extend real-time ORB endsystems.

Our earlier work on CORBA and TAO explored several dimensions of real-time ORB endsystem design including real-time scheduling [10], real-time request demultiplexing [21], real-time event processing [13], and real-time I/O subsystem integration [19]. This paper focuses on a previously unexamined point in the real-time ORB endsystem design space: *software architectures that significantly reduce priority inversion and non-determinism in CORBA ORB Core implementations*.

An ORB Core is the component in the CORBA reference model that manages transport connections, delivers client requests to an Object Adapter, and returns responses (if any) to clients. The ORB Core also typically implements the transport endpoint demultiplexing and concurrency architecture used by applications. Figure 1 illustrates how an ORB Core interacts with other CORBA components. Appendix A describes each

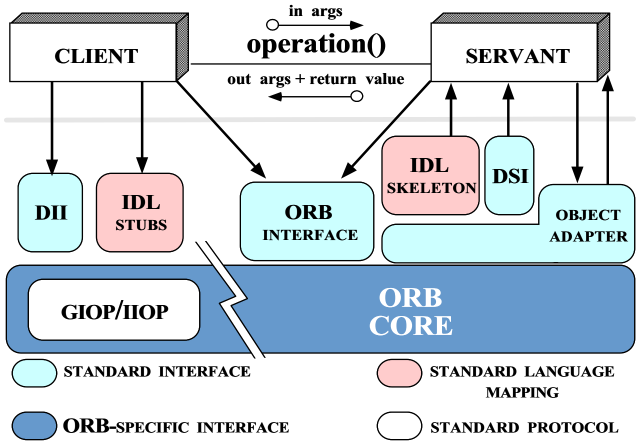

Figure 1: Components in the CORBA Reference Model

of these components in more detail.

This paper is organized as follows: Section 2 outlines the general factors that impact real-time ORB endsystem performance and predictability; Section 3 describes software architectures for real-time ORB Cores, focusing on alternative ORB Core concurrency and connection architectures; Section 4 presents empirical results from systematically measuring the efficiency and predictability of alternative ORB Core architectures in four contemporary CORBA implementations: CORBAplus, miniCOOL, MT-Orbix, and TAO; Section 5 compares our research with related work; and Section 6 presents concluding remarks.

2 Factors Impacting Real-time ORB Endsystem Performance

Meeting the QoS needs of next-generation distributed applications requires much more than defining IDL interfaces or adding preemptive real-time scheduling into an OS. It requires a vertically and horizontally integrated *ORB endsystem architecture* that can deliver end-to-end QoS guarantees at multiple levels throughout a distributed system. The key levels in an ORB endsystem include the network adapters, OS I/O subsystems, communication protocols, ORB middleware, and higher-level services.

The main thrust of this paper is on software architectures that are suitable for real-time ORB Cores. For completeness, Section 2.1 briefly outlines the general sources of overhead in ORB endsystems. Section 2.2 then describes the key sources of priority inversion and non-determinism that affect real-time ORB endsystems. After this overview, Section 3 focuses specifically on alternative ORB Core concurrency and connection architectures.

2.1 General Sources of ORB Endsystem Overhead

Our prior experience [21, 22, 23] measuring the throughput and latency of CORBA ORBs indicated that the performance overhead of real-time ORB endsystems stems from inefficiencies in the following components:

1. Network connections and network adapters: These endsystem components handle heterogeneous network connections and bandwidths, which can significantly affect latencies and cause variability in performance. Inefficient design of network adapters can cause queueing delays and lost packets [24], which are unacceptable in many real-time systems.

2. Communication protocol implementations and integration with the I/O subsystem and network adapters: Inefficient protocol implementations and improper integration with I/O subsystems can adversely affect endsystem performance. Specific factors that cause problems include the protocol overhead caused by flow control, congestion control, retransmission strategies, and connection management. Likewise, lack of proper I/O subsystem integration yields excessive data copying, fragmentation, reassembly, context switching, synchronization, checksumming, demultiplexing, marshaling, and demarshaling overhead [25].

3. ORB transport protocol implementations: Inefficient implementations of ORB transport protocols such as the CORBA Internet inter-ORB protocol (IIOP) [5] and Simple Flow Protocol (SFP) [26] can cause performance overhead and priority inversions. Specific factors responsible for these inversions include improper connection management strategies, inefficient sharing of endsystem resources, and excessive synchronization overhead in ORB protocol implementations.

4. ORB Core implementations and integration with OS services: The design of an ORB Core can yield excessive memory accesses, cache misses, heap allocations/deallocations, and context switches [27]. In turn, these factors can increase latency and jitter, which is unacceptable for distributed systems with deterministic real-time requirements. Specific factors that can cause problems include: data copying, fragmentation/reassembly, context switching, synchronization, checksumming, socket demultiplexing, timer handling, request demultiplexing, marshaling/demarshaling, framing, error checking, connection and concurrency architectures. Many of these problems are similar to those listed in bullet 2 above. Because they occur at the user-level rather than at the kernel-level, however, it can be easier for ORB implementers to solve them portably.

Figure 2 pinpoints where these various factors impact ORB performance and where optimizations can be applied to reduce key sources of ORB endsystem overhead, priority inversion, and non-determinism. Below, we focus on the sources of overhead in ORB endsystems that are chiefly responsible for priority inversions and non-determinism.

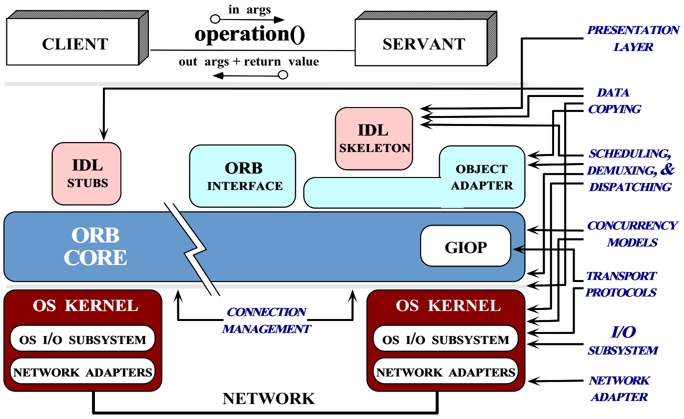

Figure 2: Optimizing Real-time ORB Endsystem Performance

2.2 Sources of Priority Inversion and Non-determinism in ORB Endsystems

Sources of priority inversion and non-determinism in ORB endsystems generally stem from resources that are shared by multiple threads or processes. Common examples of shared ORB endsystem resources include (1) TCP connections used by CORBA IIOP, (2) threads used to transfer requests through client and server end-points, (3) process-wide dynamic memory managers, and (4) internal ORB data structures like connection tables and socket/request demultiplexing maps. Below, we describe key sources of priority inversion and non-determinism in conventional ORB endsystems.

2.2.1 I/O Subsystem

The I/O subsystems of general-purpose operating systems, such as Solaris and Windows NT, do not perform preemptive, *prioritized* protocol processing [19]. In particular, the protocol processing of lower priority packets is *not* deferred due to the arrival of higher priority packets. Instead, incoming packets are processed by their arrival order rather than by their priority.

For instance, if a low-priority request arrives immediately before a high priority request, the I/O subsystem will process the lower priority packet and pass it to an application servant before the higher priority packet. The time spent in the low-priority servant represents the degree of ORB priority inversion.

[19] examines key issues that cause priority inversion in I/O subsystems and describes how TAO's real-time I/O subsystem avoids priority inversion by co-scheduling pools of user-level and kernel-level real-time threads. Interestingly, the results in Section 4 illustrate that the majority of the overhead, priority inversion, and non-determinism in ORB endsystems does *not* stem from the I/O subsystem but instead from the software architecture of the ORB Core.

2.2.2 ORB Core

A CORBA ORB Core implements the general inter-ORB protocol (GIOP) [5], which defines a standard format for interoperating between (potentially heterogeneous) ORBs. ORB Core mechanisms establish connections and implement the concurrency architecture to process GIOP requests. The following discussion outlines common sources of priority inversion and non-determinism in conventional ORB Core implementations.

Connection architecture: The ORB Core's architecture for managing connections has a major impact on real-time ORB behavior. Therefore, a key challenge for developers of real-time ORBs is to select a connection architecture that can efficiently and predictably utilize the transport mechanisms of an ORB endsystem. The following discussion outlines the key sources of priority inversion and non-determinism exhibited by conventional ORB Core connection architectures:

• **Dynamic connection management:** Conventional ORBs typically create connections dynamically in response to client requests. However, dynamic connection management can incur significant run-time overhead and priority inversion. For instance, a high-priority client may need to wait for the connection establishment of a lower-priority client. In addition, the time required to establish connections can vary widely, ranging from hundreds of microseconds to milliseconds, depending on endsystem load and network congestion.

Connection establishment overhead is hard to bound. For instance, if an ORB needs to dynamically establish connections between the client and server, it is hard to provide a reasonable guarantee of the worst-case execution time since this time also includes the (variable) connection establishment time. Moreover, connection establishment often occurs outside the scope of general end-to-end OS QoS enforcement mechanisms [28]. To support applications with deterministic real-time QoS requirements, therefore, it is generally necessary for ORB endsystems to pre-allocate connections *a priori*.

• **Connection multiplexing:** Conventional ORB Cores typically use a single TCP connection for all object references to a server process that are accessed by threads in a client process. This *connection multiplexing* is shown in Figure 3. The goal of connection multiplexing is to minimize the number of connections open to each server, which is commonly used to build scalable servers over TCP. However, connection multiplexing can yield substantial packet-level priority inversions and synchronization overhead, as shown in Sections 4.2.1 and 4.2.2.

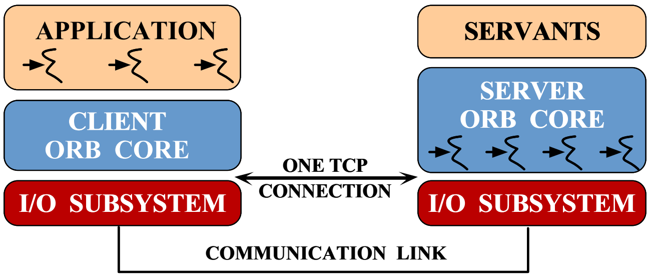

Figure 3: A Multiplexed Connection Architecture

Concurrency architecture: The ORB Core's concurrency architecture has a substantial impact on its real-time behavior. Therefore, another key challenge for developers of real-time ORBs is to select a concurrency architecture that correctly shares the aggregate processing capacity of an ORB endsystem and its application operations in one or more threads of control. The following outlines the key sources of priority inversion and non-determinism exhibited by conventional ORB Core concurrency architectures:

• **Twoway operation reply processing:** On the client-side, conventional ORB Core concurrency architectures for twoway operations can incur significant priority inversion. For instance, multi-threaded ORB Cores that use connection multiplexing incur priority inversions when low-priority threads awaiting replies from a server block out higher priority threads awaiting replies from the same server.

• **Thread pools:** On the server-side, ORB Core concurrency architectures often use *thread pools* to select a thread to process an incoming request. However, conventional ORBs do not provide programming interfaces to allow real-time applications to determine the priority of threads in this pool. Therefore, the priority of a thread in the pool is often inappropriate for the priority of the servant that ultimately executes the request, thereby increasing the potential for priority inversion.

2.2.3 Object Adapter

A standard GIOP-compliant client request contains the identity of its remote object and remote operation. A remote object is represented by an object key `octet sequence` and a remote operation is represented as a `string`. Conventional ORBs demultiplex client requests to the appropriate operation of the servant implementation using the steps shown in Figure 4.

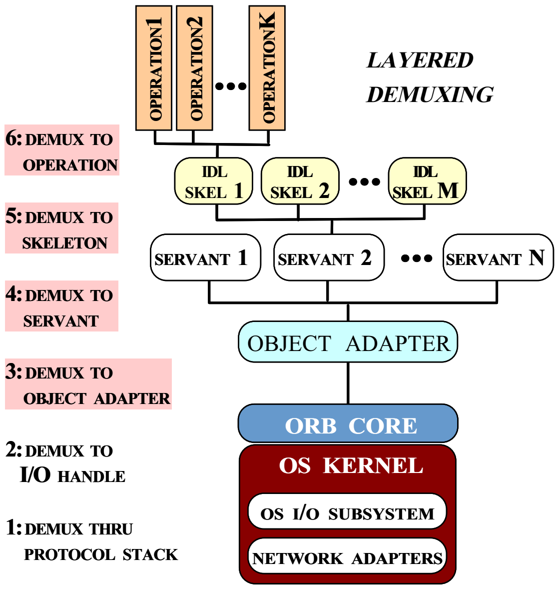

Figure 4: Layered CORBA Request Demultiplexing

These steps perform the following tasks:

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.*, through the data link, network, and transport layers up to the user/kernel boundary and the ORB core.

Steps 3, 4, and 5: The ORB core uses the addressing information in the client's object key to locate the appropriate Object Adapter, servant, and the skeleton of the target IDL operation.

Step 6: The IDL skeleton locates the appropriate operation, demarshals the request buffer into operation parameters, and performs the operation upcall.

In general, layered demultiplexing is inappropriate for high-performance and real-time applications for the following reasons [29]:

Decreased efficiency: Layered demultiplexing reduces performance by increasing the number of internal tables that must be searched while incoming client requests ascend through the processing layers in an ORB endsystem. Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an ORB.

Increased priority inversion and non-determinism: Layered demultiplexing can cause priority inversions because servant-level QoS information is inaccessible to the lowest-level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem. Therefore, the Object Adapter may demultiplex packets according to their FIFO order of arrival. FIFO demultiplexing can cause higher priority packets to wait for an indeterminate period of time while lower priority packets are demultiplexed and dispatched.

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [22, 30] show that conventional ORBs spend ~17% of the total server time processing demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide uniform quality of service guarantees to applications.

[21] presents alternative ORB demultiplexing techniques and describes how TAO's real-time Object Adapter provides optimal demultiplexing strategies that execute deterministically in constant time and avoid priority inversion via de-layered demultiplexing.

3 Alternative ORB Core Concurrency and Connection Architectures

This section describes alternative ORB Core concurrency and connection architectures. Each of these architectures is used by one or more commercial or research CORBA implementations. Below, we qualitatively evaluate how each architecture manages the aggregate processing capacity of ORB endsystem components and application operations. Section 4 then presents quantitative results that illustrate how efficient and predictable these alternatives are in practice.

3.1 Alternative ORB Core Connection Architectures

There are two general strategies for structuring connection architecture in an ORB Core: *multiplexed* and *non-multiplexed*. We describe and evaluate various design alternatives for each approach below, focusing on client-side connection architectures for our examples.

3.1.1 Multiplexed Connection Architectures

Many ORBs multiplex client requests from a single process through one TCP connection to its corresponding server process. This architecture is commonly used to build scalable ORBs by minimizing the number of TCP connections open to each server. When multiplexing is used, the key challenge is to design an efficient ORB Core connection architecture that supports concurrent reads and writes.

Multiple threads cannot portably read or write from the same socket concurrently because TCP provides untyped bytestream data transfer semantics. Therefore, concurrent write requests to a socket shared within an ORB process must be serialized. Serialization is typically implemented by having all client threads in a process acquire a lock before writing to a shared socket.

For oneway operations, there is no need for additional locking or processing once a request is sent. Implementing twoway operations over a shared connection is more complicated, however. In this case, the ORB Core must support concurrent read access to a shared socket endpoint.

If server replies are multiplexed through a single connection then multiple threads cannot read simultaneously from that socket endpoint. Instead, the ORB Core must demultiplex incoming replies to the appropriate client thread by using the GIOP sequence number sent with the original client request and returned with the servant's reply.

Several common ways of implementing connection multiplexing to allow concurrent read and write access are described below.

Active connection architecture: One approach is the *active connection* architecture shown in Figure 5. An application thread (**1**) invokes a twoway operation, which enqueues the request in the ORB (**2**). A separate thread in the ORB Core services this queue (**3**) and performs a write operation on the multiplexed socket. The ORB thread `selects`[1] (**4**) on the socket waiting for the server to reply, reads the reply from the socket (**5**), and enqueues the reply in a message queue (**6**). Finally, the application thread retrieves the reply from this queue (**7**) and returns back to its caller.

[1] The `select` call is typically used since a client may have multiple multiplexed connections to multiple servers.

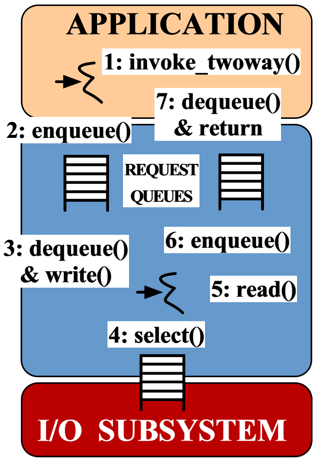

Figure 5: Active Connection Architecture

The advantage of the active connection architecture is that it simplifies the ORB connection architecture implementation by using a uniform queueing mechanism. In addition, if every socket handles packets of the same priority level, *i.e.*, packets of different priorities are not received on the same socket, the active connection can handle these packets in FIFO order without causing priority inversion.

The disadvantage with this architecture, however, is that the active connection forces an extra context switch on all outgoing/incoming operations. As a result, many ORBs use a variant of this model called the *leader/follower* connection architecture, which is described next.

Leader/followers connection architecture: An alternative to the active connection approach is the *leader/followers* architecture shown in Figure 6. As before, an application thread

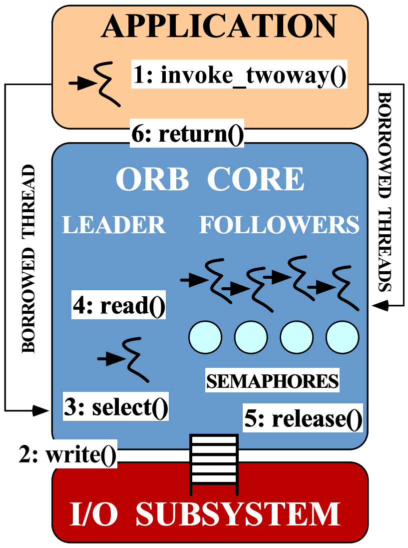

Figure 6: Leader/Follower Connection Architecture

invokes a twoway operation call (**1**). Rather than enqueueing the request in an ORB message queue, however, the request is sent across the socket immediately (**2**), using the thread of the application to perform the write. Moreover, no single thread in the ORB Core is dedicated to handling all the socket I/O in the leader/follower architecture. Instead, the first thread that attempts to wait for a reply on the multiplexed connection will block in `select` waiting for a reply (**3**). This thread is called the *leader*.

To avoid corrupting the socket bytestream, only the one leader thread can `select` on the socket(s). Thus, all client threads that "follow the leader" to read replies from the shared socket will block on semaphores managed in FIFO order by the ORB Core. If replies return from the server in FIFO order this strategy is optimal since there is no unnecessary processing or context switching. However, replies may arrive in non-FIFO order. For instance, the next reply arriving from a server could be for any one of the threads blocked on semaphores.

When the next reply arrives from the server, the leader reads the reply (**4**). It uses the sequence number returned in the GIOP reply header to identify the correct thread to receive the reply. If the reply is for the leader's own request, the leader releases the semaphore of the next follower (**5**) and returns to its caller (**6**). The next follower becomes the new leader and blocks on `select`.

If the reply is *not* for the leader, however, the leader must signal the semaphore of the appropriate thread. The signaled thread then wakes up, retrieves its reply, and returns to its caller. Meanwhile, the leader thread continues to `select` for the next reply.

Compared with active connections, the advantage of the leader/follower connection architecture is that it minimizes the number of context switches incurred *if replies arrive in FIFO order*. The drawback, however, is that the complex implementation logic can yield significant locking overhead and priority inversion. The locking overhead stems from the need to acquire mutexes when sending requests and to block on the semaphores while waiting for replies. The priority inversion occurs if the priority of the waiting threads is not considered by the leader thread when it demultiplexes replies to client threads.

3.1.2 Non-multiplexed Connection Architectures

One technique for minimizing ORB Core priority inversion is to use a non-multiplexed connection architecture, such as the one shown in Figure 7. In this connection architecture, each client thread maintains a table of pre-established connections to servers in thread-specific storage. A separate connection is maintained in each thread for every priority level, *e.g.*, P_1, P_2, P_3, etc. As a result, when a twoway operation is invoked (**1**) it shares no socket endpoints with other threads. Therefore,

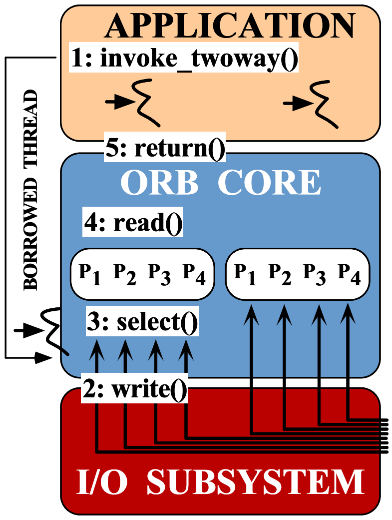

Figure 7: Non-multiplexed Connection Architecture

the write operation (**2**) and the `select` (**3**), read (**4**), and return (**5**) operations can occur without contending for ORB resources with other threads in the process.

The primary benefit of a non-multiplexed connection architecture is that it enables clients to preserve end-to-end priorities and prevent priority inversion while sending requests through ORB endsystems and across communication links. In addition, this design incurs low synchronization overhead because no additional locks are required in the ORB Core when sending/receiving twoway requests since connections are not shared.

The drawback with a non-multiplexed connection architecture is that it can use a larger number of socket endpoints than the multiplexed connection model, which may increase the ORB endsystem memory footprint. Therefore, it is most effective when used for statically configured real-time applications, such as avionics mission computing systems [19], which possess a small, fixed number of connections.

3.2 Alternative ORB Core Concurrency Architectures

There are several strategies for structuring the concurrency architecture in an ORB Core. The most common design for real-time ORBs is some variant of *thread pool*. This architecture spawns a pool of threads to service incoming client requests. In this subsection, we describe and evaluate several alternative thread pool designs, focusing largely on server-side concurrency architectures.

3.2.1 Worker Thread Pool Architecture

This ORB concurrency architecture uses a design similar to the active connection architecture described in Section 3.1.1. The structure of this design is illustrated in Figure 8. The primary components in this design include an I/O thread, a request queue, and a pool of worker threads. The I/O thread `selects` (**1**) on the socket endpoints, reads (**2**) new client requests, and (**3**) inserts them into the tail of the request queue. A worker thread in the pool dequeues (**4**) the next request from the head of the queue and dispatches it (**5**).

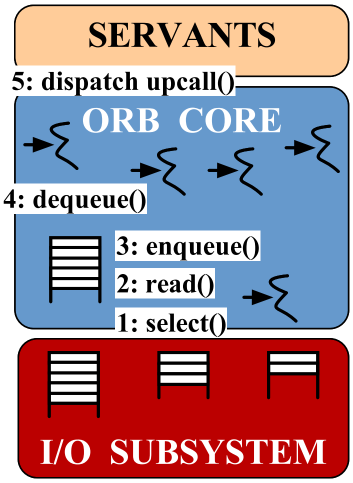

Figure 8: Server-side Worker Thread Pool Concurrency Architecture

The chief advantage of the worker thread pool concurrency architecture is that it is straightforward to implement. The disadvantages of this model stem from the excessive context switching and synchronization required to manage the request queue, as well as priority inversion caused due to connection multiplexing. Since different priority requests share the same transport connection, a high priority request may wait until a lower priority request that arrived earlier is processed. Moreover, additional priority inversions can occur if the priority of the thread that dispatches the request is different than the priority of the servant that processes the request.

3.2.2 Leader/Follower Thread Pool Architecture

This ORB concurrency architecture is an optimization of the worker thread pool model. Its design, which is similar to the leader/follower connection architecture discussed in Section 3.1.1 is shown in Figure 9. A pool of threads is allocated and a leader thread is chosen to `select` (**1**) on connections for all servants in the server process. When a request arrives, this thread reads (**2**) it into an internal buffer. If this is a valid request for a servant, a follower thread in the pool is released to become the new leader (**3**) and the leader thread dispatches the upcall (**4**). After the upcall is dispatched, the original leader thread becomes a follower and returns to the thread pool. New requests are queued in socket endpoints until a thread in the pool is available to execute the requests.

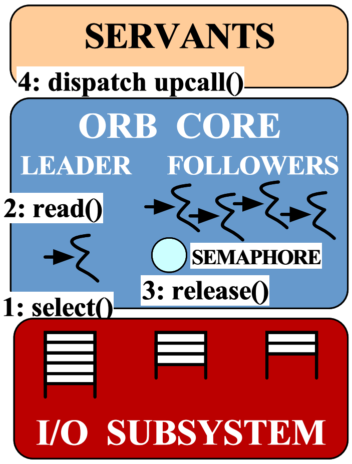

Figure 9: Server-side Leader/Follower Concurrency Architecture

Compared with the worker thread design, the chief advantage of the leader/follower concurrency architecture is that it minimizes context switching overhead incurred by incoming requests. This is because it is not necessary to transfer the request from the thread that read it from the socket endpoint to another thread in the pool that processes it. The disadvantages of the leader/follower architecture are the same as with the worker thread design.

3.2.3 Threading Framework Architecture

A more flexible way to implement an ORB concurrency architecture is to allow application developers to customize hooks provided by a general *threading framework*. One way of structuring this approach is shown in Figure 10. The design in this figure is based on the MT-Orbix thread filter concurrency framework, which is a variant of the Chain of Responsibility pattern [16]. In MT-Orbix, an application can install a thread filter at the top of a chain of filters. Filters are application-programmable hooks that can perform a number of tasks such as intercepting, modifying, or examining each request sent to and from the ORB.

A thread in the ORB Core reads (**1**) a request from a socket endpoint and enqueues the request on a request queue in the ORB Core (**2**). Another thread then dequeues the request (**3**) and passes it through each filter in the chain successively. The topmost filter (*i.e.*, the thread filter) determines which thread should handle this request. In the *thread-pool* model, the thread filter enqueues the request into a queue serviced by a

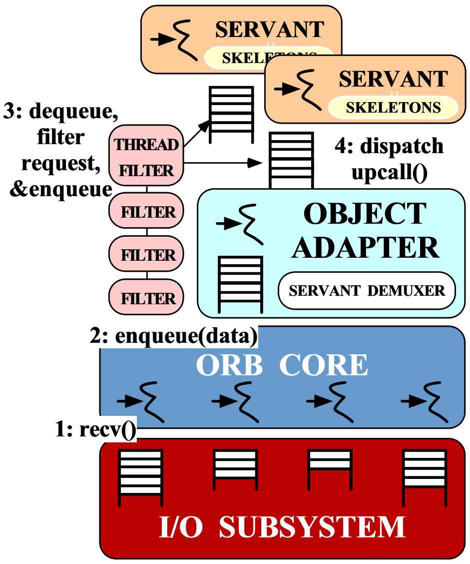

Figure 10: Server-side Thread Pool Framework Concurrency Architecture

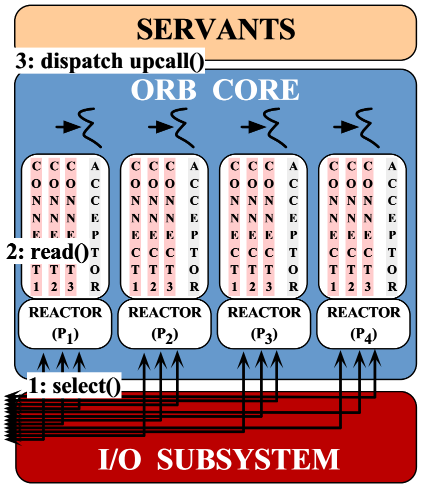

Figure 11: Server-side Thread-per-Priority Concurrency Architecture

thread with the appropriate priority. This thread then passes control back to the ORB, which performs operation demultiplexing and dispatches the upcall (**4**).

The main advantage of a threading framework is its flexibility. The thread filter mechanism can be programmed by server applications to support various concurrency strategies. For instance, to implement a *thread-per-request* concurrency policy, the filter can spawn a new thread and pass the request to this new thread.[2]

There are several disadvantages with this design, however. First, since there is only a single chain of filters, extensive priority inversion can occur since each request must traverse the filter chain in FIFO order. Second, there may be FIFO queueing at multiple levels in the ORB endsystem. Therefore, a high priority request may only be processed after several lower priority requests that arrived earlier. Third, the threading framework may increase locking overhead, *e.g.*, the thread filter must acquire locks to enqueue requests into the queue of the appropriate thread.

3.2.4 Thread-per-Priority Thread Pool Architecture

In this approach, the server associates each servant with a thread using the thread-per-priority concurrency architecture shown in Figure 11. The ORB Core can be configured to pre-allocate a real-time thread for each priority level. For instance, avionic mission computing systems commonly execute their tasks in fixed priority threads corresponding to the *rates* (*e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz) at which operations are called by clients.

To minimize context switching, each thread in the ORB Core can be configured with a `Reactor` [31]. A `Reactor` demultiplexes (**1**) all incoming client requests to the appropriate connection handler, *i.e.*, connect_1, connect_2, etc. The connection handler reads (**2**) the request and dispatches (**3**) it to a servant that execute at its thread priority.

Each `Reactor` in a server is also associated with an `Acceptor` [32]. The `Acceptor` is a factory that listens on a particular port number for clients to connect to that thread priority and creates a connection handler to process the GIOP requests. In the example in Figure 11, there is a listener port per priority. Thus, ports 10020, 10010, 10005, 10001 correspond to the 20 Hz, 10 Hz, 5 Hz, and 1 Hz rate group thread priorities, respectively.

The advantage of the thread-per-priority concurrency architecture is that it minimizes priority inversion and non-determinism. Moreover, it reduces context switching and synchronization overhead by only locking the state of servants if they interact across different thread priorities. In addition, this concurrency model supports scheduling and analysis techniques that associate priority with rate, such as Rate Monotonic Scheduling (RMS) and Rate Monotonic Analysis (RMA) [33, 34].

The thread-per-priority concurrency model can be integrated seamlessly with the non-multiplexed connection model described in Section 3.1.2 to provide end-to-end priority preservation in real-time ORB endsystems, as shown in Figure 12. Once a client connects, the `Acceptor` creates a new socket queue and connection handler to service that queue. The I/O subsystem uses the port number contained in arriv-

[2]The thread-per-request architecture is generally unsuited for real-time applications since the overhead of creating a thread for each request is excessive and non-deterministic.

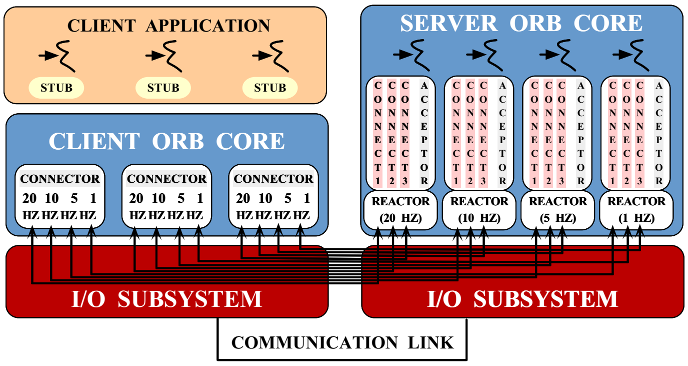

Figure 12: End-to-end Real-time ORB Core Software Architecture

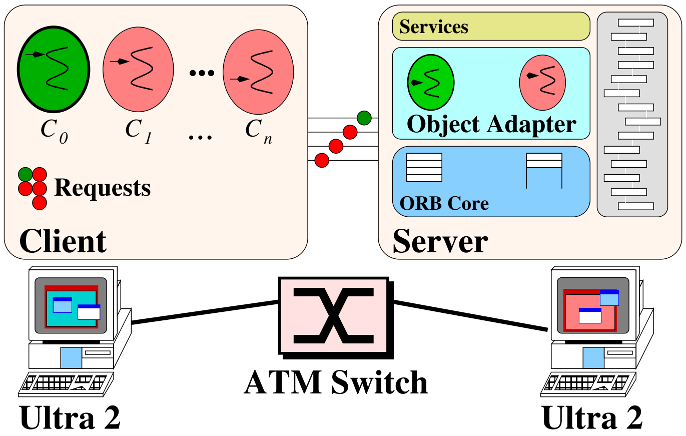

Figure 13: Testbed for ORB Endsystem Evaluation

ing requests as a demultiplexing key to associate requests with the appropriate socket queue. This design minimizes priority inversion through the entire distributed ORB endsystem by eagerly demultiplexing [11] incoming requests onto the appropriate real-time thread that services the priority level of the target servant.

4 Real-time ORB Core Performance Experiments

This section describes the results of experiments that measure the real-time behavior of several commercial and research ORBs, including IONA's MT-Orbix 2.2, Sun miniCOOL 4.3[3], Expersoft CORBAplus 2.1.1 and TAO 1.0. MT-Orbix and CORBAplus are not real-time ORBs, *i.e.*, they were not explicitly designed to support applications with real-time QoS requirements. Sun miniCOOL is a subset of the COOL ORB that is specifically designed for embedded systems with small memory footprints. TAO was designed at Washington University to support real-time applications with deterministic and statistical quality of service requirements, as well as best effort requirements.

4.1 Benchmarking Testbed

This section describes the experimental testbed we designed to systematically measure sources of latency and throughput overhead, priority inversion, and non-determinism in ORB endsystems. The architecture of our testbed is depicted in Figure 13. The hardware and software components in the experiments are described briefly below.

[3] COOL was previously developed by Chorus, which was recently acquired by Sun.

4.1.1 Hardware Configuration

The experimental testbed is depicted in Figure 13. The experiments were conducted using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The LattisCell 10114 is a 16-Port, OC3 155 Mbps/port switch. Each UltraSPARC-2 contains 2 168 MHz CPUs[4] with a 1 Megabyte cache per-CPU, 256 Megabytes of RAM, and an ENI-155s-MF ATM adaptor card that supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

4.1.2 Client/Server Configuration and Benchmarking Methodology

Server benchmarking configuration: As shown in Figure 13, our testbed server consists of two servants within the Object Adapter. One servant runs in a higher priority thread than the other. Each thread processes requests that are sent to its servant by client threads on the other UltraSPARC-2.

Solaris real-time threads [35] are used to implement servant priorities. The high-priority servant thread has the *highest* real-time priority available on Solaris and the low-priority servant has the *lowest* real-time priority.

The server benchmarking configuration is implemented in the various ORBs as follows:

• **CORBAplus:** which uses the worker thread pool architecture described in Section 3.2.1. In version 2.1.1. of COR-

[4] To ensure that all the real-time threads were competing for the same CPU, the second CPU was disabled using the Solaris `psradm(1M)` utility.

BAplus, by default, every multi-threaded application has at least two threads: an initial (main) thread of execution, and an event dispatching thread. The latter receives the requests and passes them on to the user threads, that processes them.

• **miniCOOL:** which uses the leader/follower thread pool architecture described in Section 3.2.2. Version 4.3 of miniCOOL allows application-level concurrency control. The application developer can choose between thread-per-request or thread-pool. The thread-pool concurrency architecture was used for our benchmarks since it is better suited than thread-per-request for deterministic real-time applications. In the thread-pool concurrency architecture, the developer initially spawns a fixed number of threads. In addition, miniCOOL dynamically spawns threads on behalf of server applications to handle requests, whenever the initial threads are insufficient, as shown in Figure 9.

• **MT-Orbix:** which uses the thread pool architecture based on the Chain of Responsibility pattern described in Section 3.2.3. The server creates two threads at startup time. The high-priority thread is associated with the high-priority servant and the low-priority thread is associated with the low-priority servant. Incoming requests are assigned to these threads using the Orbix thread filter mechanism, as shown in Figure 10. Each priority has its own queue of requests, to avoid priority inversion within the queue, which can otherwise occur if a high priority servant and a low-priority servant dequeue requests from the same queue.

• **TAO:** which uses the thread-per-priority concurrency architecture described in Section 3.2.4. Version 1.0 of TAO integrates the thread-per-priority concurrency architecture with the non-multiplexed connection architecture, as shown in Figure 12. In contrast, the other three ORBs multiplex all client requests over a single connection to the server.

Client benchmarking configuration: Figure 13 shows how the benchmarking test used one high-priority client C_0 and n low-priority clients, $C_1 \ldots C_n$. The high-priority client runs in a high-priority real-time OS thread and invokes operations at 20 Hz, *i.e.*, it invokes 20 CORBA twoway calls per second. The low-priority clients run in lower-priority OS threads[5] and invoke operations at 10 Hz, *i.e.*, they invoke 10 CORBA twoway calls per second. In each call, the client sends a value of type `CORBA::Octet` to the servant. The servant cubes the number and returns it to the client.

When the test program creates the client threads, they block on a barrier lock so that no client begins work until the others are created and ready to run. When all threads inform the main thread they are ready to begin, the main thread unblocks all client threads, which then execute in an arbitrary order determined by the Solaris real-time thread dispatcher. Each client invokes 4,000 CORBA twoway requests at the prescribed rate.

[5] All low-priority clients have the same OS thread priority.

4.2 Performance Results

Two categories of tests were used in our benchmarking experiments: *blackbox* and *whitebox*.

Blackbox benchmarks: We computed the average twoway response time incurred by various clients. In addition, we computed twoway operation jitter, which is the standard deviation from the average twoway response time. High levels of latency and jitter are undesirable for deterministic real-time applications since they complicate the computation of worst-case execution time and reduce CPU utilization. Section 4.2.1 explains the blackbox results.

Whitebox benchmarks: To precisely pinpoint the *source* of priority inversion and performance non-determinism, we employed whitebox benchmarks. These benchmarks used profiling tools such as UNIX `truss(1)` and `Quantify` [36]. These tools trace and log the activities of the ORBs and measure the time spent on various tasks, as explained in Section 4.2.2.

Together, the blackbox and whitebox benchmarks indicate the end-to-end latency/jitter incurred by CORBA clients and help explain the reason for these results, respectively. In general, the results reveal why ORBs like MT-Orbix, CORBAplus, and miniCOOL are not yet suited for applications with deterministic real-time performance requirements. Likewise, the results illustrate empirically how and why the ORB Core architecture used by TAO is more suited for these types of applications.

4.2.1 Blackbox Results

As the number of low-priority clients increases, the number of low-priority requests sent to the server also increases. Ideally, a real-time ORB endsystem should show no variance in the latency observed by the high-priority client, irrespective of the number of low-priority clients. However, our measurements of end-to-end twoway ORB latency yielded the results in Figure 14. This figure also shows that as the number of low-priority clients increases, MT-Orbix and CORBAplus incur significantly higher latencies, *i.e.*, 7 times as much as TAO. In addition, the MT-Orbix and miniCOOL low priority clients exhibit very high levels of jitter, *i.e.*, 100 times as much as TAO in the worst case, as shown in Figure 15.

The blackbox results for each ORB are explained below.

CORBAplus results: The excessive use of user-level locks in CORBAplus, as shown in Figure 24, caused it to incur the

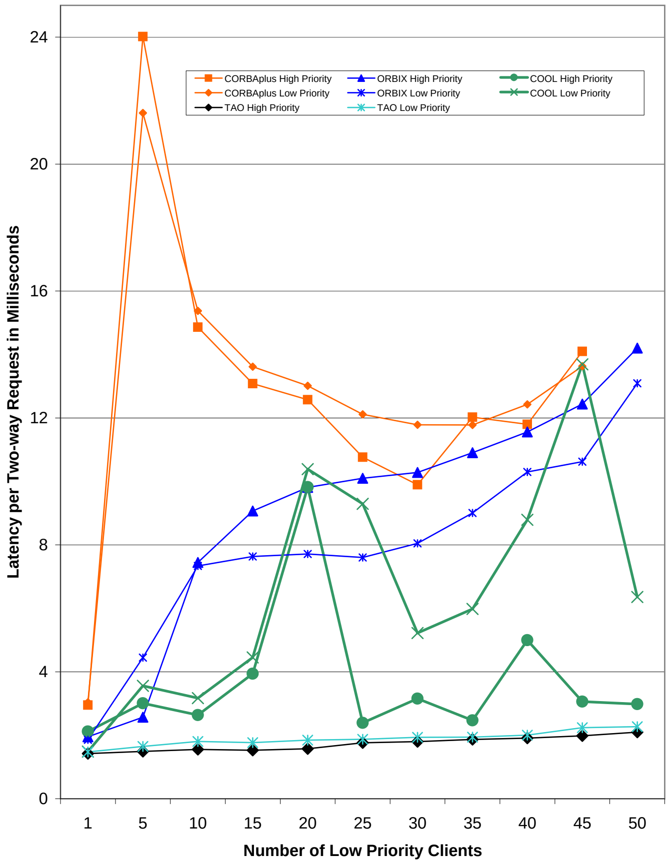

Figure 14: Comparative Latency for CORBAplus, MT-Orbix, miniCOOL, and TAO

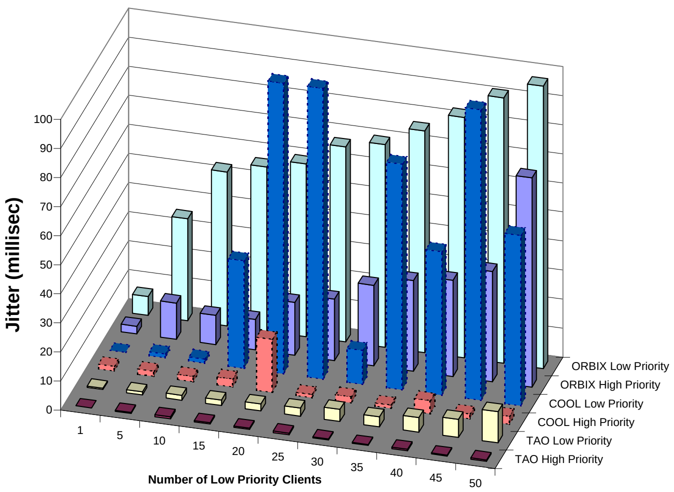

Figure 15: Comparative Jitter for MT-Orbix, miniCOOL and TAO

highest overhead of the ORBs we tested. Moreover, CORBAplus incurs priority inversion at various points in the graph. After displaying a high amount of latency for a small number of low-priority clients, the latency drops suddenly at 10 clients, then rises gradually. Clearly, this behavior is unsuitable for deterministic real-time applications. Section 4.2.2 reveals how the poor performance and priority inversions stem largely from CORBAplus' concurrency architecture.[6]

MT-Orbix results: MT-Orbix incurs substantial priority inversion as the number of low-priority clients increase. After the number of clients exceeds 10, the high-priority client performs increasingly worse than the low-priority clients. Clearly, this behavior is not conducive to deterministic real-time applications. Section 4.2.2 reveals how these inversions stem largely from the MT-Orbix concurrency architecture on the server. In addition, the MT-Orbix ORB produces high levels of jitter, as shown in Figure 15. This behavior is caused by priority inversions in its ORB Core, as explained in Section 4.2.2.

miniCOOL results: As the number of low-priority clients increase, the latency observed by the high-priority client increases, reaching ∼10 msec, at 20 clients, at which point it decreases suddenly to 2.5 msec at the 25 client round. This erratic behavior becomes more evident as the number of low-priority clients increase. Although the latency of the high-priority client is smaller than the low-priority clients, the non-linear behavior of the clients makes miniCOOL unsuitable for deterministic real-time applications.

The difference in latency between the high- and the low-priority client is also non-deterministic. For instance, it evolves from 0.55 msec to 10 msec. Section 4.2.2 reveals how this behavior stems largely from the connection architecture used by the miniCOOL client and server.

The jitter incurred by miniCOOL is also fairly high, as shown in Figure 15. This jitter is not as high as that observed with the MT-Orbix ORB, however, since miniCOOL's concurrency architecture does not perform as much locking overhead or use as many FIFO queues.

TAO results: Figure 14 reveals that as the number of low-priority clients increases from 1 to 50, the latency observed by TAO's high-priority client grows by ∼0.7 msecs. However, the difference between the low and the high priority clients starts at 0.05 msec and ends at 0.27 msec. In contrast, in miniCOOL, it evolves from 0.55 msec to 10 msec. Also, TAO's rate of increase is significantly lower than both MT-Orbix and Sun miniCOOL. In particular, when there are 50 low-priority clients competing for the CPU and network bandwidth, the latency observed with MT-Orbix is more than 7 times that of

[6] Note to reviewers: due to bugs with the latest version of CORBAplus, jitter results for this ORB are not yet available. We plan to include them in Figure 15 for the final version of this paper.

TAO and the miniCOOL latency is ∼3 times that of TAO in the low priority clients.

TAO's high-priority client always performs better than its lower priority clients. This indicates that connection and concurrency architectures in TAO's ORB Core are well suited for maintaining real-time request priorities end-to-end. The key difference between TAO and the other ORBs are that TAO's GIOP protocol processing is performed on a dedicated connection by a dedicated real-time thread with a suitable end-to-end real-time priority. Thus, TAO shares the minimal amount of ORB endsystem resources, which substantially reduces opportunities for priority inversion and overhead.

The TAO ORB produces very low jitter (less than 11 msecs) for the low-priority requests and negligible jitter (less than 1 msec) for the high-priority requests. The stability of TAO's latency is clearly desirable for applications that require predictable end-to-end performance. In addition, these results illustrate that improper choice of ORB Core concurrency and connection software architectures can play a larger role in exacerbating priority inversion and non-determinism than the I/O subsystem.

4.2.2 Whitebox Results

For the whitebox tests, we used a configuration of ten concurrent clients similar to the one described in Section 4.1. Nine clients were low-priority and one was high-priority. Each client sent 4,000 twoway requests to the server, which had a low-priority servant and high-priority servant thread.

Our previous performance studies suggested that locks constitute a significant source of overhead, non-determinism and potential priority inversion for real-time ORBs. Using `Quantify` and `truss`, we measured the time consumed by the ORBs performing tasks like synchronization, I/O, and protocol processing. In addition, we computed a metric that records the number of calls made to user-level locks (*i.e.*, `mutex_lock` and `mutex_unlock`) and kernel-level locks (*i.e.*, `_lwp_mutex_lock`, `_lwp_mutex_unlock`, `_lwp_sema_post` and `_lwp_sema_wait`). This metric computes the average number of lock operations per request. In general, kernel-level locks are considerably more expensive since they incur mode switching overhead.

These whitebox results are presented below.

CORBAplus whitebox results: Our whitebox analysis reveals that synchronization overhead from mutex and semaphore operations at the user-level consume a large percentage of the total CORBAplus ORB processing time, as shown in Figure 24. Synchronization overhead arises from mutex and semaphore locking operations that implement the connection and concurrency architecture used by CORBAplus.

As shown in Figure 16 CORBAplus displays synchronization overhead using kernel-level locks in the client side.[7]

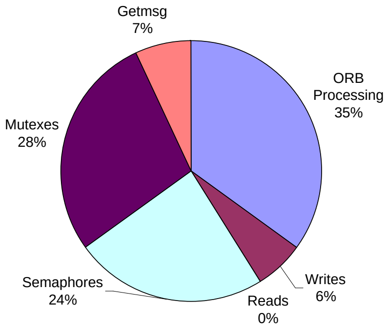

Figure 16: Client-side Whitebox Results for CORBAplus

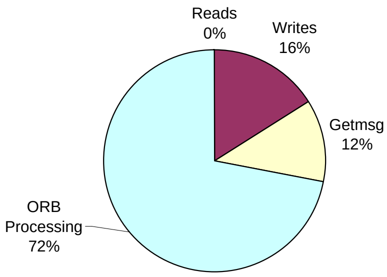

Figure 17: Server-side Whitebox Results for CORBAplus

For each CORBA request/response, CORBAplus's client ORB performs 199 lock operations, whereas the server performs 216 user-level lock operations. This locking overhead stems largely from excessive dynamic memory allocation, as described in Section 4.3. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

The CORBAplus connection and concurrency architectures are outlined briefly below.

- **CORBAplus connection architecture:** The CORBAplus ORB connection architecture uses a simple model of the active connection architecture described in Section 3.1.1 and depicted in Figure 8. This design multiplexes all requests through one TCP connection.

- **CORBAplus concurrency architecture:** The CORBAplus ORB concurrency architecture uses the thread pool architecture described in Section 3.2.1 and depicted in Figure 8. This architecture uses a single I/O thread to accept and

[7]Note to reviewers: due to bugs with the latest version of CORBAplus, overhead from locks in the server side for CORBAplus are not yet available. We plan to include them in Figure 17 for the final version of this paper.

read requests from socket endpoints. This thread enqueues the request on a queue that is serviced by a pool of worker threads.

The CORBAplus connection architecture and the server concurrency architecture work well to reduce the number of simultaneous open connections and simplify the implementation. However, concurrent requests to the shared connection incur high-levels of synchronization and context switching, as well as cause priority inversion. For instance, on the client-side, threads of different priorities can share the same transport connection. Therefore, a high-priority thread may be blocked until a lower priority thread finishes sending its request. In addition, the priority of the thread that blocks on the semaphore to receive a reply from a twoway connection may not reflect the priority of the *request* that arrives from the server, thereby causing additional priority inversion.

miniCOOL whitebox results: Our whitebox analysis reveals that synchronization overhead from mutex and semaphore operations consume a large percentage of the total miniCOOL ORB processing time. Synchronization overhead arises from mutex and semaphore locking operations that implement the connection and concurrency architecture used by miniCOOL.

Locking overhead accounted for ∼50% on the client-side (shown in Figure 18) and more than 40% on the server-side (shown in Figure 19).

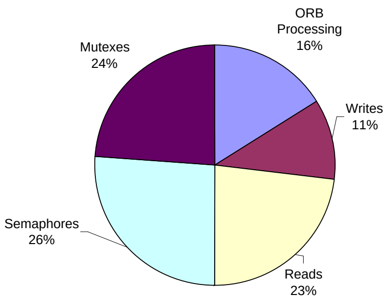

Figure 18: Client-side Whitebox Results for miniCOOL

For each CORBA request/response, miniCOOL's client ORB performs 94 lock operations at the user-level, whereas the server performs 231 lock operations, as shown in Figure 24. As with CORBAplus, this locking overhead stems from excessive dynamic memory allocation. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

In addition, the number of calls per request to kernel-level locking mechanisms at the server, (shown in Figure 25) are unusually high, due to the fact that miniCOOL uses "bound" threads on Solaris, which require kernel intervention for all synchronization operations.

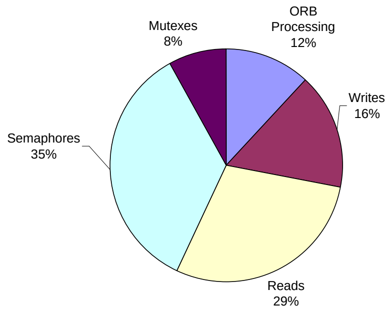

Figure 19: Server-side Whitebox Results for miniCOOL

The miniCOOL connection and concurrency architectures are outlined briefly below.

- **miniCOOL connection architecture:** The miniCOOL ORB connection architecture uses a variant of the leader/followers architecture described in Section 3.1.1. This architecture allows the first thread to perform the read on the shared socket, *i.e.*, the leader blocks in `read`. All following threads block on semaphores waiting for one of two conditions: (1) the leader thread will read their reply message and signal their semaphore or (2) the leader thread will read its own reply and signal another thread to enter and block in `read`, thereby becoming the new leader.

Thus, miniCOOL multiplexes multiple object references in one client process to a server process through a single connection. This leader/follower connection architecture minimizes the number of simultaneous connections. However, miniCOOL's connection architecture also increases overhead and potential for priority inversion. These problems arise since connection multiplexing requires multiple threads to read/write to a single socket connection shared by the threads.

- **miniCOOL concurrency architecture:** The Sun miniCOOL ORB concurrency architecture uses the leader/followers thread pool architecture described in Section 3.2.2. This architecture initially uses a single thread to wait for connections. Whenever a request arrives and validation of the request is complete, the leader thread (1) signals a follower thread in the pool to wait for incoming requests and (2) services the request.

The miniCOOL connection architecture and the server concurrency architecture help reduce the number of simultaneous open connections and the amount of context switching when replies arrive in FIFO order. However, this design yields high levels of priority inversion. For instance, threads of different priorities can share the same transport connection on the client-side. Therefore, a high-priority thread may block until a lower priority thread finishes sending its request. In addition, the priority of the thread that blocks on the semaphore to access a connection may not reflect the priority of the *response* that arrives from the server, which yields additional priority inversion.

MT-Orbix whitebox results: Figure 20 shows the whitebox results for the client-side and Figure 21 shows the whitebox results for the server-side of MT-Orbix.

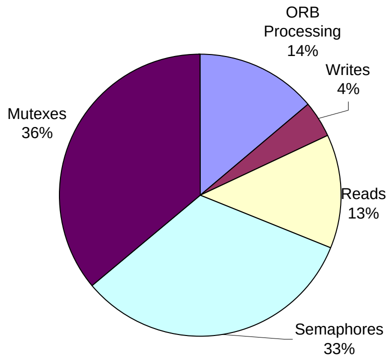

Figure 20: Client-side Whitebox Results for MT-Orbix

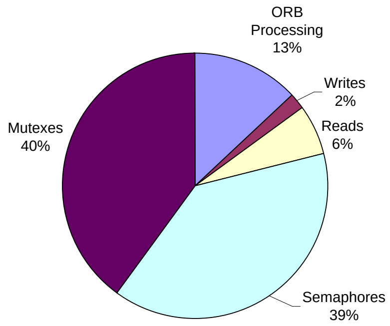

Figure 21: Server-side Whitebox Results for MT-Orbix

• **MT-Orbix connection architecture:** Like miniCOOL, MT-Orbix uses the leader/follower connection architecture, described in Section 3.2.2. Although this model minimizes context switching overhead, it causes intensive priority inversions, as explained in Section 3.2.2.

• **MT-Orbix concurrency architecture:** In the MT-Orbix implementation of our benchmarking testbed, multiple servant threads were created, each with the appropriate priority, *i.e.*, high-priority servants had a high-priority thread. A thread filter was then installed to look at each request, determine the priority of the request (by examining the target object), and pass the request to the thread with the correct priority. The thread filter mechanism is implemented by a high-priority real-time thread to minimize the dispatch time.

The thread pool instantiation of the MT-Orbix mechanism described in Section 3.2.3 is flexible and easy to use. However, it suffers from high levels of priority inversion and synchronization overhead. The MT-Orbix ORB provides only *one* thread filter chain. Therefore, all incoming requests must be sequentially processed by the filter before they are passed to the servant thread with an appropriate real-time priority. As a result, if a high-priority request arrives after a low-priority request, it must wait until the low-priority request has been dispatched before it can be processed.

In addition, a filter can only be called after (1) IIOP processing has completed and (2) the Object Adapter has determined the target object for this request. This ORB processing is serialized since the MT-Orbix protocol engine is unaware of the request priority. Thus, a higher priority request that arrived after a low-priority request must wait until the lower priority request has been processed by the ORB Core.

The concurrency architecture is chiefly responsible for the substantial priority inversion exhibited by MT-Orbix, as shown in Figure 14. This figure shows how the latency observed by the high-priority client increases rapidly, from $\sim$2 msecs to $\sim$14 msecs as the number of low-priority clients increase from 1 to 50.

In addition, the MT-Orbix filter mechanism causes an increase in synchronization overhead. Because there is just one filter chain, concurrent requests must acquire and release locks to be processed by the filter. The MT-Orbix client-side performs 175 user-level lock operations per request, while the server-side performs 599 user-level lock operations per request, as shown in Figure 24. Moreover, MT-Orbix also displays a high number of kernel-level locks per request as shown in Figure 25.

TAO whitebox results: As shown in Figures 22 and 23, TAO exhibits negligible synchronization overhead. TAO performs 41 user-level lock operations per request on the client-side, and 100 user-level lock operations per request on the server-side. This low amount of synchronization results from the design of TAO's ORB Core, which allocates a separate connection for each priority, as shown in Figure 12. Therefore, TAO's ORB Core minimizes additional user-level locking operations per request and uses no kernel-level locks in its ORB Core.

• **TAO connection architecture:** TAO uses a non-multiplexed connection architecture, which pre-establishes

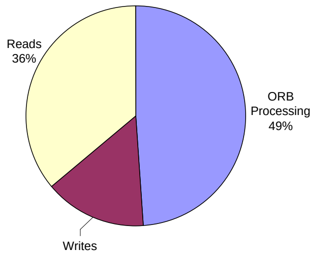

Figure 22: Client-side Whitebox Results for TAO

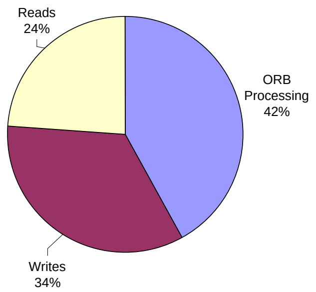

Figure 23: Server-side Whitebox Results for TAO

connections to servants, as described in Section 3.1.2. One connection is pre-established per priority level, thereby avoiding the non-deterministic delay involved in dynamic connection setup. In addition, different priority levels have their own connection, thus avoiding priority inversion due to the FIFO ordering of packet transmission by the network and I/O subsystem.

• **TAO concurrency architecture:** TAO supports a variety of concurrency architectures, as described in [19]. The *thread-per-priority* architecture was used for the benchmarks described in this paper. In this concurrency architecture, a separate thread is created for each priority level *i.e.*, each rate group. Thus, the low-priority client issues CORBA requests at a lower rate (10 Hz) than the high-priority client (20 Hz).

On the server-side, client requests sent to the high-priority servant are processed by a high-priority real-time thread. Likewise, client requests sent to the low-priority servant are handled by the low-priority real-time thread. Locking overhead is minimized since these two servant threads share minimal ORB resources. In addition, the two threads service separate client connections, thereby eliminating the priority inversion that otherwise arises from connection multiplexing, as exhibited by the other ORBs we tested.

Locking overhead: Our whitebox tests measured user-level locking overhead (shown in Figure 24) and kernel-level locking overhead (shown in Figure 25) in the CORBAplus, MT-Orbix, miniCOOL and TAO ORBs. User-level locks are typically used to protect shared resources within a process. A common example is dynamic memory allocation since memory is allocated from a global per-process heap.

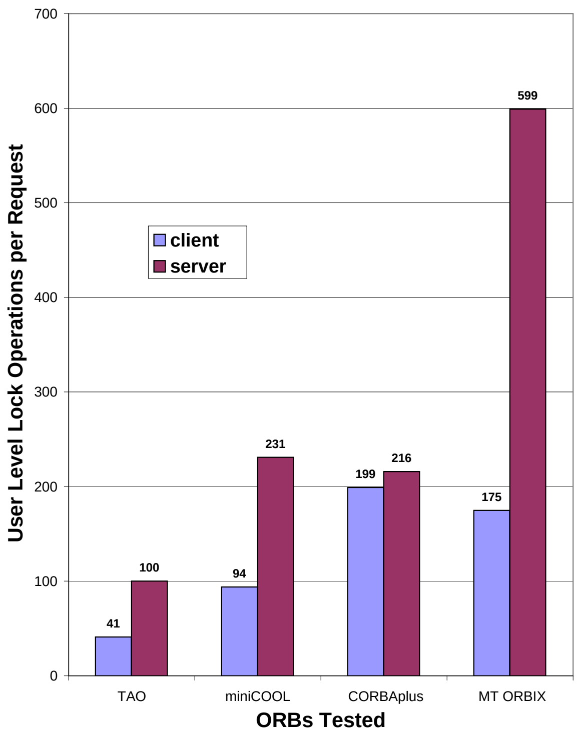

Figure 24: User-level Locking Overhead in ORBs

Kernel-level locks are more expensive since they typically require mode switches between user-level and the kernel. The semaphore and mutex operations depicted in the whitebox results for the ORBs arise from kernel-level lock operations.

TAO limits user-level locking by using pre-allocated buffers. A single buffer is allocated per request. This buffer is subdivided to accommodate the various fields of the request. Kernel-level locking is limited due to the fact that ORB resources are not shared between the threads.

4.3 Evaluation and Recommendations

The results of our benchmarks illustrate the non-deterministic performance incurred by applications running atop conventional ORBs. In addition, the results show that priority inver-

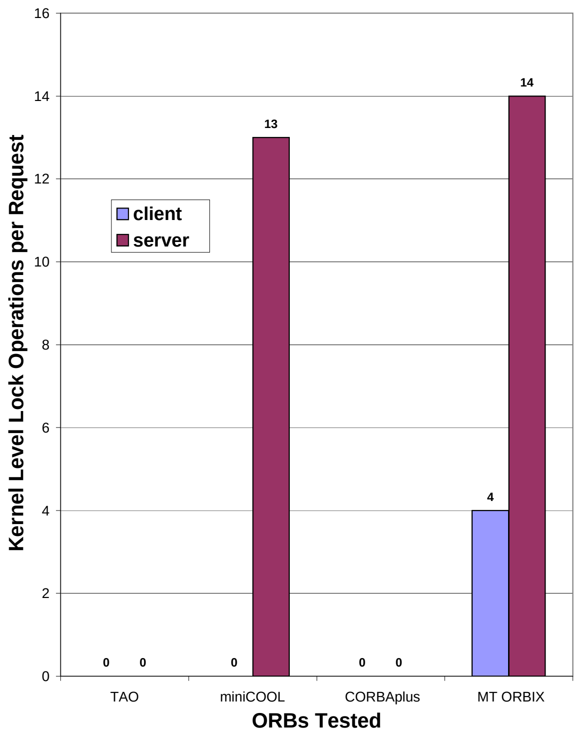

Figure 25: Kernel-level Locking Overhead in ORBs

sion and non-determinism are significant problems in conventional ORBs. As a result, these ORBs are currently unsuitable for applications with deterministic real-time requirements. Based on our results, and our past experience [21, 22, 23, 30] measuring the performance of CORBA ORB endsystems, we suggest the following recommendations to decrease non-determinism and limit priority inversion in real-time ORB endsystems.

1. Real-time ORBs should avoid dynamic connection establishment: ORBs that establish connections dynamically suffer from high jitter. Thus, performance seen by individual clients can vary significantly from the average. Neither CORBAplus, miniCOOL, nor MT-Orbix provide APIs for pre-establishing connections, though TAO does provide these APIs as extensions to CORBA.

We recommend that APIs to control the pre-establishment of connections should be defined as an OMG standard.

2. Real-time ORBs should avoid multiplexing requests of different priorities over a shared connection: Sharing connections requires synchronization. Thus, high-priority requests can be blocked until low-priority threads release the shared connection lock.

We recommend that real-time ORBs should allow application developers to determine whether requests with different priorities are multiplexed over shared connections. Currently, neither miniCOOL, CORBAplus, nor MT-Orbix supports this level of control, though TAO provides this flexibility.

3. Real-time ORBs should minimize dynamic memory allocation: Thread-safe implementations of dynamic memory allocators require user-level locking. For instance, the C++ `new` operator allocates memory from a global pool shared by all threads in a process. Likewise, the C++ `delete` operation, that releases allocated memory, also requires user-level locking to update the global shared pool. This lock sharing contributes to the overhead shown in Figure 24.

We recommend that real-time ORBs avoid excessive sharing of dynamic memory locks via the use of OS features such as thread-specific storage [37], which allocates memory from heaps that are unique in each thread.

4. Real-time ORB concurrency architectures should be flexible, yet efficient and predictable: Many ORBs, such as miniCOOL and CORBAPlus, create threads on behalf of server applications. This design prevents application developers from customizing ORB performance by selecting an appropriate concurrency architecture. Conversely, other ORB concurrency architectures are flexible, but inefficient and non-deterministic, as shown in the Section 4.2.2 explanation of the MT-Orbix performance results. Thus, a balance is needed between flexibility and efficiency.

We recommend that real-time ORBs provide APIs that allow application developers to select concurrency architectures that are flexible, efficient, *and* predictable. For instance, TAO offers a range of concurrency architectures (such as thread-per-priority, thread pool, and thread-per-connection) that are carefully designed using thread-specific storage to minimize unnecessary sharing of ORB resources.

5. The real-time ORB endsystem architecture should be guided by empirical performance benchmarks: Our prior research on pinpointing performance bottlenecks and optimizing middleware like Web servers [38, 39] and CORBA ORBs [22, 21, 30, 23] demonstrates the efficacy of this measurement-driven research methodology.

We recommend that the OMG adopt standard real-time CORBA benchmarking techniques and metrics. These benchmarks will simplify the communication and comparison of performance results and real-time ORB behavior patterns.

5 Related Work

An increasing number of research efforts are focusing on integrating QoS into CORBA. The work presented in this paper is based on the TAO project [10]. This section compares TAO with related work.

Krupp, *et al*, at MITRE Corporation were among the first to elucidate the needs of real-time CORBA systems [40]. They identified key requirements and outlined mechanisms for supporting end-to-end timing constraints [41]. A system consisting of a commercial off-the-shelf RTOS, a CORBA-compliant ORB, and a real-time object-oriented database management system is under development [42]. Similar to the TAO approach, the initial static scheduling approach is rate monotonic, but a strategy for dynamic deadline monotonic scheduling support has been designed [41]. Other dynamic scheduling approaches may be considered in the future.

Wolfe, *et al*, are developing a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) [43]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed operation invocations (`TDMIs`) [44]. A `TDMI` corresponds to TAO's `RT_Operation` [19] and an `RT_Environment` structure contains QoS parameters similar to those in TAO's `RT_Info` [10].

One difference between TAO and the URI approaches is that `TDMIs` [41] express required timing constraints, *e.g.*, deadlines relative to the current time, whereas TAO's `RT_Operations` publish their resource, *e.g.*, CPU time, requirements. The difference in approaches may reflect the different time scales, seconds versus milliseconds, respectively, and scheduling requirements, dynamic versus static, of the initial application targets. However, the approaches should be equivalent with respect to system schedulability and analysis.

The QuO project at BBN [45] has defined a model for communicating changes in QoS characteristics between applications, middleware, and the underlying endsystems and network. The QuO model uses the concept of a connection between a client and an object to define QoS characteristics, and treats these characteristics as first-class objects. These objects can then be aggregated to enable the characteristics to be defined at various levels of granularity, *e.g.*, for a single method invocation, for all method invocations on a group of objects, and similar combinations. The model also uses 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.

The QuO architecture differs from our work on real-time QoS provision since QuO does not provide hard real-time guarantees of ORB endsystem CPU scheduling. Furthermore, the QuO programming model involves the use of several QDL specifications, in addition to OMG IDL, based on the separation of concerns advocated by Aspect-Oriented Programming (AOP) [46]. We believe that while the AOP paradigm is quite powerful, the proliferation of definition languages may be overly complex for common application use-cases. Therefore, the TAO programming model focuses on the `RT_Operation` and `RT_Info` QoS specifiers, which can be expressed in standard OMG IDL.

The Epiq project [47] defines an open real-time CORBA scheme that provides QoS guarantees and runtime scheduling flexibility. Epiq extends TAO's off-line scheduling model to provide on-line scheduling. In addition, Epiq allows clients to be added and removed dynamically via an admission test at runtime. The Epiq project is work-in-progress and does not yet have empirical results.

The ARMADA project [48] defines a set of communication and middleware services that supports fault-tolerant and end-to-end guarantees for real-time distributed applications. ARMADA provides real-time communication services based on the X-kernel and the Open Group's MK microkernel. This infrastructure serves as a foundation for constructing higher-level real-time middleware services. TAO differs from ARMADA in that most of the real-time features in TAO are built using TAO's ORB Core. In addition, TAO implements the OMG's CORBA standard, while also providing the hooks that are necessary to integrate with an underlying real-time I/O subsystem. Thus, the real-time services provided by ARMADA's communication system can be utilized by TAO's ORB Core to support a vertically integrated real-time system.

6 Concluding Remarks

Conventional CORBA ORBs exhibit substantial priority inversion and non-determinism. Consequently, they are not yet suited for distributed, real-time applications with deterministic QoS requirements. Meeting these demands requires that ORB Core software architectures be designed to reduce priority inversion. The TAO ORB Core described in this paper minimizes priority inversion by using a priority-based concurrency architecture and non-multiplexed connection architecture that share a minimal amount of resources among ORB Core threads. The architectural principles used in TAO can be applied to other ORBs and other real-time software systems.

TAO has been used to develop a real-time ORB endsystem for avionics mission computing applications. These applications manage sensors and operator displays, navigate the aircraft's course, and control weapon release. To meet the scheduling demands of mission computing applications, TAO supports real-time scheduling and dispatching of periodic processing operations, as well as efficient event filtering and correlation mechanisms [13]. The C++ source code for TAO and ACE is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. This release also contains the real-time ORB benchmarking test suite described in Section 4.1.

Acknowledgments

We gratefully acknowledge Expersoft, IONA, and Sun for providing us with their ORB software for the benchmarking testbed. In addition, we would like to thank Frank Buschmann for extensive comments on this paper.

References

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

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

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

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

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

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

[7] Object Management Group, *Minimum CORBA - Request for Proposal*, OMG Document orbos/97-06-14 ed., June 1997.

[8] Object Management Group, *Realtime CORBA 1.0 Request for Proposals*, OMG Document orbos/97-09-31 ed., September 1997.

[9] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[10] 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.

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

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

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

[14] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[15] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

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

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

[18] A. Gokhale and D. C. Schmidt, "Design Principles and Optimizations for High-performance ORBs," in 12^{th} OOPSLA Conference, poster session, (Atlanta, Georgia), ACM, October 1997.

[19] D. C. Schmidt, R. Bector, D. Levine, S. Mungee, and G. Parulkar, "An ORB Endsystem Architecture for Statically Scheduled Real-time Applications," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[20] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," *Communications of the ACM, to appear*, 1998.

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

[22] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[23] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, (London, England), pp. 50–56, IEEE, November 1996.

[24] Z. D. Dittia, J. Jerome R. Cox, and G. M. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," in *IEEE INFOCOM '95*, (Boston, USA), pp. 179–187, IEEE Computer Society Press, April 1995.

[25] N. C. Hutchinson and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[26] Object Management Group, *Control and Management of A/V Streams Request For Proposals*, OMG Document telecom/96-08-01 ed., August 1996.

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

[28] W. R. Stevens, *TCP/IP Illustrated, Volume 2*. Reading, Massachusetts: Addison Wesley, 1993.

[29] D. L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[30] A. Gokhale and D. C. Schmidt, "Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks," in *Proceedings of the International Conference on Distributed Computing Systems*, (Baltimore, Maryland), IEEE, May 1997.

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

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

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

[34] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1993.

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

[36] P. S. Inc., *Quantify User's Guide*. PureAtria Software Inc., 1996.

[37] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently," in *The* 4^{th} *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

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

[39] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[40] B. Thuraisingham, P. Krupp, A. Schafer, and V. Wolfe, "On Real-Time Extensions to the Common Object Request Broker Architecture," in *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Experiences with CORBA*, ACM, Oct. 1994.

[41] G.Cooper, L. C. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thurasignham, S. Wohlever, and V. F. Wolfe, "Real-Time CORBA Development at MITRE, NRaD, Tri-Pacific and URI," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[42] "Statement of Work for the Extend Sentry Program, CPFF Project, ECSP Replacement Phase II," Feb. 1997. Submitted to OMG in response to RFI ORBOS/96-09-02.

[43] 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.

[44] 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.

[45] 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.

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

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

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

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

A Overview of the CORBA ORB Reference Model

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

• **Object location:** CORBA objects can be located locally with the client or remotely on a server, without affecting their implementation or use;

• **Programming language:** The languages supported by CORBA include C, C++, Java, Ada95, and Smalltalk, among others.

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

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

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

The components in the CORBA reference model shown in Figure 26 provide the transparency described above. The components in CORBA include the following:

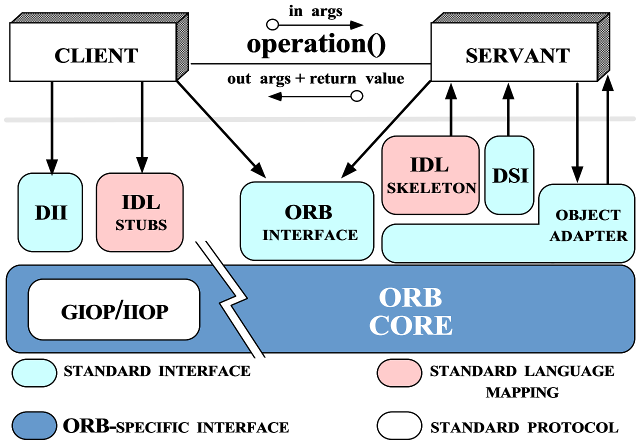

Figure 26: Components in the CORBA Reference Model

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

Client: This program entity performs application tasks by obtaining object references to servants and invoking operations on the servants. Servants can be remote or co-located relative to the client. Ideally, accessing a remote servant should be as simple as calling an operation on a local object, *i.e.,* `object->operation(args)`. Figure 26 shows the components that ORBs use to transmit requests transparently from client to servant for remote operation invocations.

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

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

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as the "glue" between the client and servants, respectively, and the ORB. Stubs provide a strongly-typed, static invocation interface (SII) that marshals application data into a common packet-level representation. Conversely, skeletons demarshal the packet-level representation back into typed data that is meaningful to an application. An IDL compiler automatically transforms OMG IDL definitions into an application programming language like C++ or Java. IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [49].

Dynamic Invocation Interface (DII): The DII allows a client to access the underlying request transport mechanisms provided by the ORB Core. The DII is useful when an application has no compile-time knowledge of the interface it is accessing. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of twoway operations to avoid blocking the client until the servant responds. In contrast, SII stubs only support twoway (*i.e.*, request/response) and oneway (*i.e.*, request only) operations.

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

Object Adapter: An Object Adapter associates a servant with an ORB, demultiplexes incoming requests to the servant, and dispatches the appropriate operation upcall on that servant. While current CORBA implementations are typically limited to a single Object Adapter per ORB, recent CORBA portability enhancements [17] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB.