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

Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan
{schmidt,sumedh,sergio}@cs.wustl.edu
Washington University
Campus Box 1045
St. Louis, MO 63130

Aniruddha Gokhale†
gokhale@research.bell-labs.com
Bell Labs, Lucent Technologies
600 Mountain Ave Rm 2A-442
Murray Hill, NJ 07974

This paper was published in the Kluwer Journal of Real-time Systems, to appear, 1999.

Abstract

There is increasing demand to extend Object Request Broker (ORB) middleware to support distributed applications with stringent real-time requirements. However, conventional ORB implementations, such as CORBA ORBs, 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 evaluates connection and concurrency software architectures to identify strategies 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 to support certain types of real-time applications over the Internet.

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

1 Introduction

Next-generation distributed real-time applications, such as video conferencing, avionics mission computing, and process control, 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 trend in real-time R&D projects away from developing real-time applications from scratch to *integrating* applications using reusable components based on object-oriented (OO) middleware [3]. The objective of middleware is to increase quality and decrease the cycle-time and effort required to develop software by supporting the integration of reusable components implemented by different suppliers.

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 deterministic and statistical QoS requirements. These components must be customizable to meet the functionality and QoS requirements of applications developed in diverse contexts.

Increased focus on standardizing and leveraging real-time COTS hardware and software: To leverage development effort and reduce training, porting, and maintenance costs, there is increasing demand to exploit the rapidly advancing capabilities of standard common-off-the-shelf (COTS) hardware and COTS operating systems. Several international standardization efforts are currently addressing QoS-related issues associated with COTS hardware and software.

One particularly noteworthy standardization effort has yielded the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification [5]. CORBA is OO middleware software 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 CORBA for real-time [7] and embedded [8] systems. Several OMG groups, most notably the Real-Time Special Interest

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

†Work done by the author while at Washington University.

Group (RT SIG), are defining standard extensions to CORBA to support distributed real-time applications. The goal of standardizing real-time CORBA is to enable real-time applications to interwork throughout embedded systems and heterogeneous distributed environments, such as the Internet.

However, developing, standardizing, and leveraging distributed real-time Object Request Broker (ORB) middleware remains hard, notwithstanding the significant efforts of the OMG RT SIG. There are few successful examples of standard, widely deployed distributed real-time ORB middleware running on COTS operating systems and COTS hardware. Conventional CORBA ORBs are generally unsuited for performance-sensitive, distributed real-time applications due to their (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) overall 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 solutions [10]. Moreover, relatively little systems research has focused on strategies and tactics for real-time ORB endsystems. For instance, QoS research at the network and OS layers is only beginning to address key requirements and programming models of ORB middleware [11].

Historically, research on QoS for high-speed networks, such as ATM, has focused largely on policies for allocating virtual circuit bandwidth [12]. Likewise, research on real-time operating systems has focused largely on avoiding priority inversions in synchronization and dispatching mechanisms for multi-threaded applications [13]. An important open research topic, therefore, is to determine how best to map the results from QoS work at the network and OS layers onto the OO programming model familiar to many real-time applications developers who use ORB middleware.

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 software 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. For completeness, Appendix A provides an overview of the CORBA reference model.

2 Factors Impacting Real-time ORB Endsystem Performance

Meeting the QoS needs of next-generation distributed applications requires much more than defining Interface Definition Language (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 [10]. The key levels in an ORB endsystem include the network adapters, OS I/O subsystems, communication protocols, ORB middleware, and higher-level services shown in Figure 1.

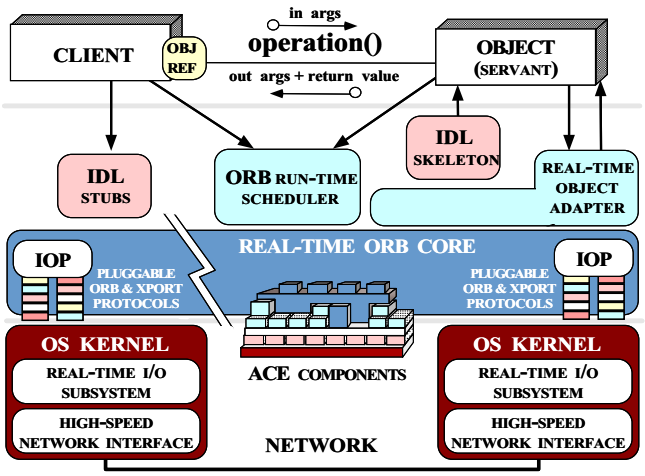

Figure 1: Components in the TAO Real-time ORB Endsystem

The main focus of this paper is on software architectures that are suitable for real-time ORB Cores. The 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 the standard CORBA components in more detail.

For completeness, Section 2.1 briefly outlines the general sources of performance overhead in ORB endsystems. Section 2.2 describes the key sources of priority inversion and non-determinism that affect the predictability and utilization of real-time ORB endsystems. After this overview, Section 3 explores alternative ORB Core concurrency and connection architectures.

2.1 General Sources of ORB Endsystem Performance Overhead

Our experience [14, 15, 16, 17] measuring the throughput and latency of CORBA implementations indicates that performance overheads in real-time ORB endsystems arise from inefficiencies in the following components:

1. Network connections and network adapters: These endsystem components handle heterogeneous network connections and bandwidths, which can significantly increase latency and cause variability in performance. Inefficient design of network adapters can cause queueing delays and lost packets [18], which are unacceptable for certain types of 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 inefficiencies 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 [19].

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) [20], can cause significant performance overhead and priority inversion. 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: An improperly designed ORB Core can yield excessive memory accesses, cache misses, heap allocations/deallocations, and context switches [21]. In turn, these factors can increase latency and jitter, which is unacceptable for distributed applications with deterministic real-time requirements. Specific ORB Core factors that cause inefficiencies 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 inefficiencies are similar to those listed in bullet 2 above. Since they occur at the user-level rather than at the kernel-level, however, ORB implementers can often address them more readily.

Figure 2 pinpoints where the various factors outlined above impact ORB performance and where optimizations can be applied to reduce key sources of ORB endsystem overhead, priority inversion, and non-determinism. Below, we describe the components in an ORB endsystem that are chiefly responsible for priority inversion and non-determinism.

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

Minimizing priority inversion and non-determinism is important for real-time operating systems and ORB middleware in order to bound application execution times. In ORB endsystems, priority inversion and non-determinism generally stem from resources that are shared between multiple threads or processes. Common examples of shared ORB endsystem resources include (1) TCP connections used by a CORBA IIOP protocol engine, (2) threads used to transfer requests through client and server transport endpoints, (3) process-wide dynamic memory managers, and (4) internal ORB data structures like connection tables for transport endpoints and demultiplexing maps for client requests. Below, we describe key sources of priority inversion and non-determinism in conventional ORB endsystems.

2.2.1 The OS I/O Subsystem

An I/O subsystem is the component in an OS responsible for mediating ORB and application access to low-level network and OS resources, such as device drivers, protocol stacks, and the CPU(s). Key challenges in building a high-performance, real-time I/O subsystem are (1) to minimize context switching and synchronization overhead and (2) to enforce QoS guarantees while minimizing priority inversion and non-determinism [22].

A context switch is triggered when an executing thread relinquishes the CPU it is running on voluntarily or involuntarily. Depending on the underlying OS and hardware platform, a context switch may require hundreds of instructions to flush register windows, memory caches, instruction pipelines, and translation look-aside buffers [23]. Synchronization overhead arises from locking mechanisms that serialize access to shared resources like I/O buffers, message queues, protocol connection records, and demultiplexing maps used during protocol processing in the OS and ORB.

The I/O subsystems of general-purpose operating systems, such as Solaris and Windows NT, do not perform preemptive, prioritized protocol processing [24]. Therefore, 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.

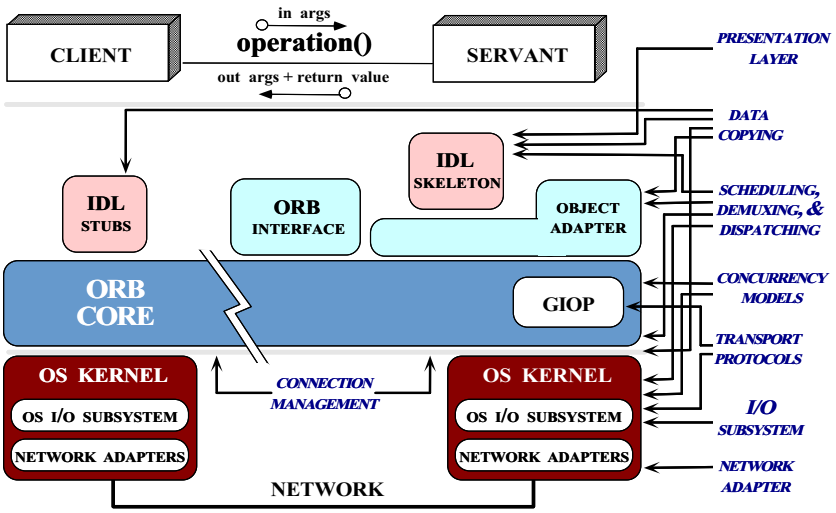

Figure 2: Optimizing Real-time ORB Endsystem Performance

For instance, in Solaris 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 priority inversion incurred by the ORB endsystem and application.

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

2.2.2 The ORB Core

An ORB Core is the component in CORBA that implements the General Inter-ORB Protocol (GIOP) [5], which defines a standard format for interoperating between (potentially heterogeneous) ORBs. The ORB Core establishes connections and implements concurrency architectures that 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 *connection architecture*, which defines how requests are mapped onto network 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 utilize the transport mechanisms of an ORB endsystem efficiently and predictably. 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 create connections dynamically in response to client requests. Dynamic connection management can incur significant run-time overhead and priority inversion, however. 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 microseconds to milliseconds, depending on endsystem load and network congestion.

  Connection establishment overhead is difficult to bound. For instance, if an ORB needs to dynamically establish connections between a client and a server, it is hard to provide a reasonable guarantee of the worst-case execution time since this time must include the (often variable) connection establishment time. Moreover, connection establishment often occurs outside the scope of general end-to-end OS QoS protocol enforcement mechanisms, such as retransmission timers [25]. To support applications with deterministic real-time QoS requirements, therefore, ORB endsystems often must preallocate connections *a priori*.

- **Connection multiplexing:** Conventional ORB Cores typically share a single multiplexed TCP connection for all object references to servants in 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, *e.g.*, to improve server scalability 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.

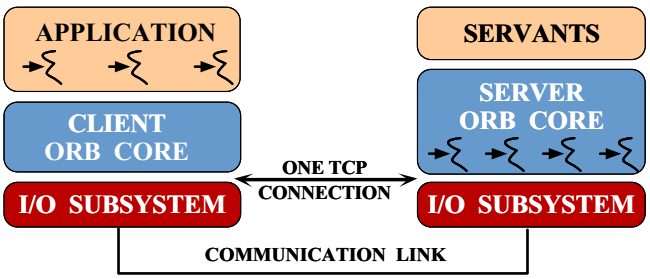

Figure 3: A Multiplexed Connection Architecture

Concurrency architecture: The ORB Core's *concurrency architecture*, which defines how requests are mapped onto threads, also 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 can effectively share the aggregate processing capacity of an ORB endsystem and its application operations in one or more threads. The following outlines the key sources of priority inversion and non-determinism exhibited by conventional ORB Core concurrency architectures:

- **Two-way operation reply processing:** On the client-side, conventional ORB Core concurrency architectures for two-way 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* [26] to select a thread in which to process an incoming request. However, conventional ORBs do not provide programming interfaces that allow real-time applications to assign 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. An improperly designed ORB Core increases the potential for, and duration of, priority inversion and non-determinism [27].

2.2.3 The Object Adapter

An Object Adapter is the component in CORBA that is responsible for demultiplexing incoming requests to servant operations that handle the request. A standard GIOP-compliant client request contains the identity of its object and operation. An object is identified by an object key, which is an `octet sequence`. An operation is represented as a `string`. As shown in Figure 4, the ORB endsystem must perform the following demultiplexing tasks:

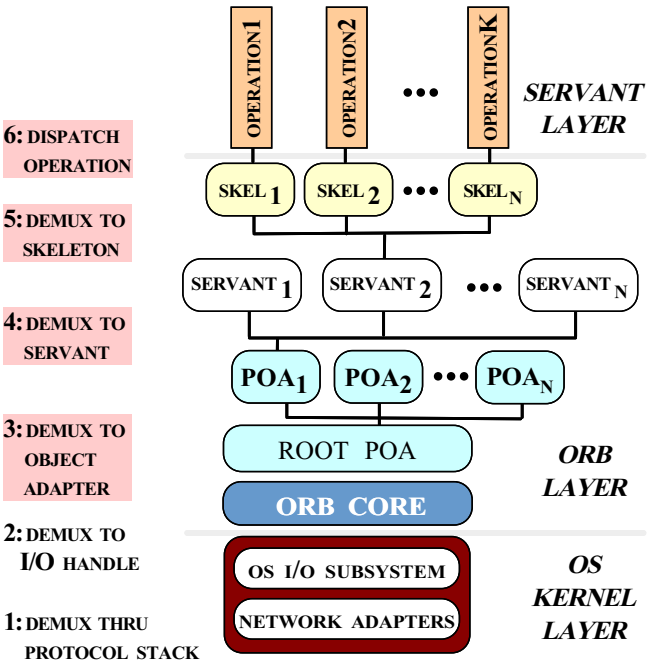

Figure 4: CORBA 2.2 Logical Server Architecture

Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request multiple times, starting from the network interface, through the data link, network, and transport layers up to the user/kernel boundary (*e.g.*, the socket layer), where the data is passed to the ORB Core in a server process.

Steps 3, and 4: The ORB Core uses the addressing information in the client's object key to locate the appropriate POA and servant. POAs can be organized hierarchically. Therefore, locating the POA that contains the designated servant can involve a number of demultiplexing steps through the nested POA hierarchy.

Step 5 and 6: The POA uses the operation name to find the appropriate IDL skeleton, which demarshals the request buffer into operation parameters and performs the upcall to code supplied by servant developers to implement the object's operation.

The conventional deeply-layered ORB endsystem demultiplexing implementation shown in Figure 4 is generally inappropriate for high-performance and real-time applications for the following reasons [28]:

Decreased efficiency: Layered demultiplexing reduces performance by increasing the number of internal tables that must be searched as 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 Object Adapter.

Increased priority inversion and non-determinism: Layered demultiplexing can cause priority inversions because servant-level quality of service (QoS) information is inaccessible to the lowest-level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem. Therefore, an Object Adapter may demultiplex packets according to their FIFO order of arrival. FIFO demultiplexing can cause higher priority packets to wait for a non-deterministic period of time while lower priority packets are demultiplexed and dispatched [22].

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [15, 17] 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, scalable QoS guarantees to real-time applications.

[14] presents alternative ORB demultiplexing techniques and describes how TAO's real-time Object Adapter provides optimal demultiplexing strategies that execute deterministically in constant time. The demultiplexing strategies used in TAO also avoid priority inversion via *de-layered demultiplexing*, which removes unnecessary layering within TAO's Object Adapter.

3 Alternative ORB Core Concurrency and Connection Architectures

This section describes a number of common ORB Core concurrency and connection architectures. Each architecture 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 efficiently and predictably these alternatives perform 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 in our examples.

3.1.1 Multiplexed Connection Architectures

Most conventional ORBs multiplex all client requests emanating from threads in a single process through one TCP connection to their corresponding server process. This multiplexed connection architecture is used to build scalable ORBs by minimizing the number of TCP connections open to each server. When multiplexing is used, however, a key challenge is to design an efficient ORB Core connection architecture that supports concurrent `read` and `write` operations.

TCP provides untyped bytestream data transfer semantics. Therefore, multiple threads cannot `read` or `write` from the same socket concurrently. Likewise, `writes` to a socket shared within an ORB process must be serialized. Serialization is typically implemented by having a client thread acquire a lock before writing to a shared socket.

For one-way operations, there is no need for additional locking or processing once a request is sent. Implementing two-way operations over a shared connection is more complicated, however. In this case, the ORB Core must allow multiple threads to concurrently "`read`" from a shared socket endpoint.

If server replies are multiplexed through a single TCP 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` operations are described below.

Active connection architecture:

• **Overview:** One approach is the *active connection* architecture shown in Figure 5. An application thread (**1**) invokes

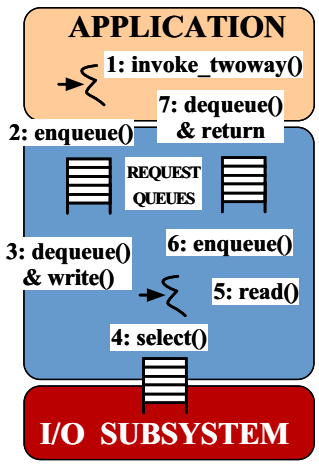

Figure 5: Active Connection Architecture

a two-way 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.

• **Advantages:** The advantage of the active connection architecture is that it simplifies ORB implementations 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 request-level priority inversion [22].

• **Disadvantages:** The disadvantage with this architecture, however, is that the active connection forces an extra context switch on all two-way operations. Therefore, to minimize this overhead, many ORBs use a variant of the active connection architecture described next.

Leader/Followers connection architecture:

• **Overview:** An alternative to the active connection model is the *leader/followers* architecture shown in Figure 6. As before, an application thread invokes a two-way operation

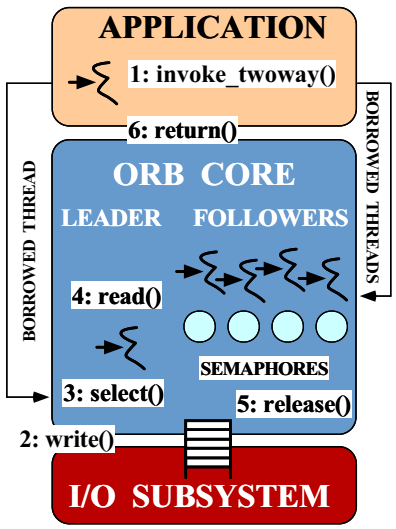

Figure 6: Leader/Follower Connection Architecture

call (**1**). Rather than enqueueing the request in an ORB message queue, however, the request is sent across the socket immediately (**2**), using the application thread that invoked the operation 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 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 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. Since replies may arrive in non-FIFO order, however, the next reply from a server could be for any one of the client 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 thread releases the semaphore of the next follower (**5**) and returns to its caller (**6**). The next follower thread becomes the new leader and blocks on `select`.

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

• **Advantages:** 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*.

• **Advantages:** The disadvantage of the leader/follower model 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 priorities of the waiting threads are not respected by the leader thread when it demultiplexes replies to client threads.

3.1.2 Non-multiplexed Connection Architectures

• **Overview:** 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 preestablished connections to servers in thread-specific storage [29]. 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 two-way operation is invoked (**1**) it shares no socket endpoints with other threads. Therefore, the `write`, (**2**), `select` (**3**),

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

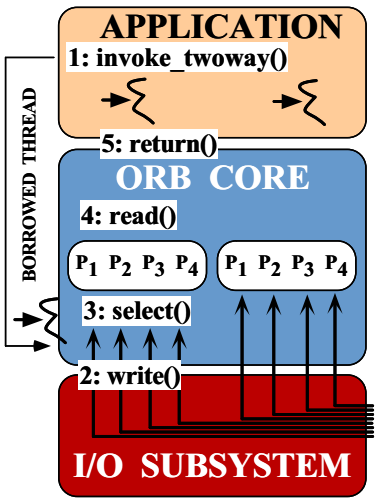

Figure 7: Non-multiplexed Connection Architecture

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

• **Advantages:** The primary advantages of a non-multiplexed connection architecture is that it preserves end-to-end priorities and minimizes priority inversion while sending requests through ORB endsystems. In addition, since connections are not shared, this design incurs low synchronization overhead because no additional locks are required in the ORB Core when sending and receiving two-way requests [30].

• **Disadvantages:** The disadvantage 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. Moreover, this approach does not scale with the number of priority levels. Therefore, it is most effective when used for statically configured real-time applications, such as avionics mission computing systems [22], which possess a small, fixed number of connections and priority levels, where each priority level maps to an OS thread priority.

3.2 Alternative ORB Core Concurrency Architectures

There are a variety of strategies for structuring the concurrency architecture in an ORB [26]. Below, we describe a number of alternative ORB Core concurrency architectures, focusing on server-side concurrency.

3.2.1 The Worker Thread Pool Architecture

• **Overview:** This ORB concurrency architecture uses a design similar to the active connection architecture described in Section 3.1.1. As shown in Figure 8, the components in

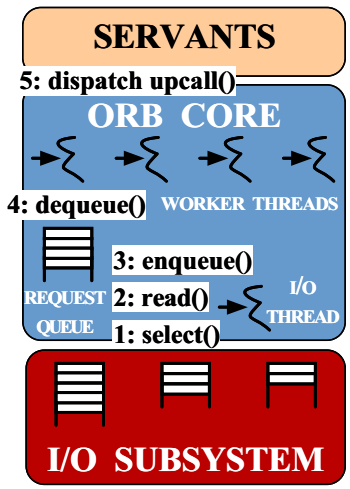

Figure 8: Server-side Worker Thread Pool Concurrency Architecture

a worker thread pool 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**).

• **Advantages:** The chief advantage of the worker thread pool concurrency architecture is its ease of implementation. In particular, the request queue provides a straightforward producer/consumer design.

• **Disadvantages:** The disadvantages of this model stem from the excessive context switching and synchronization required to manage the request queue, as well as request-level priority inversion caused by connection multiplexing. Since different priority requests share the same transport connection, a high-priority request may wait until a low-priority request that arrived earlier is processed. Moreover, thread-level priority inversions can occur if the priority of the thread that originally `reads` the request is lower than the priority of the servant that processes the request.

3.2.2 The Leader/Follower Thread Pool Architecture

• **Overview:** The leader/follower thread pool architecture is an optimization of the worker thread pool model. It is similar to the leader/follower connection architecture discussed in Section 3.1.1. As 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

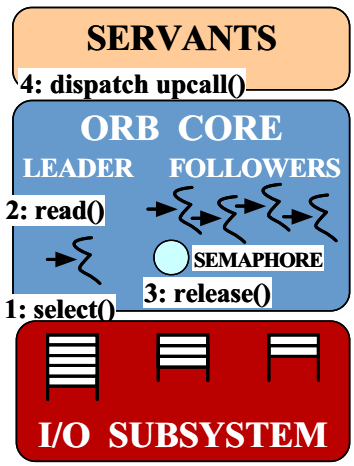

Figure 9: Server-side Leader/Follower Concurrency Architecture

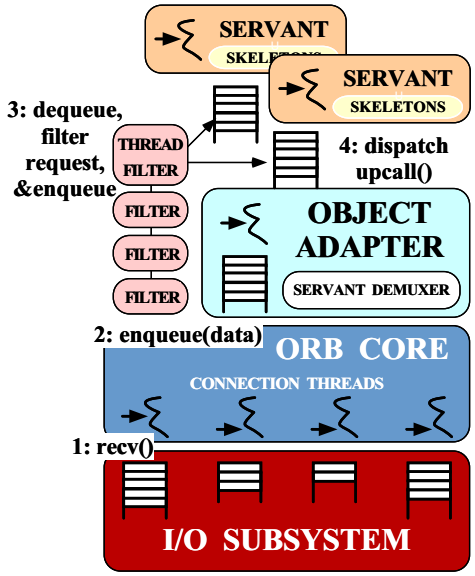

Figure 10: Server-side Thread Framework Concurrency Architecture

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.

• **Advantages:** Compared with the worker thread pool design, the chief advantage of the leader/follower thread pool architecture is that it minimizes context switching overhead incurred by incoming requests. Overhead is minimized since the request need not be transferred from the thread that read it to another thread in the pool that processes it.

• **Disadvantages:** The leader/follower architecture's disadvantages are largely the same as with the worker thread design. In addition, it is harder to implement the leader/follower model than the worker thread pool model.

3.2.3 Threading Framework Architecture

• **Overview:** A very flexible way to implement an ORB concurrency architecture is to allow application developers to customize hook methods provided by a *threading framework*. One way of structuring this framework is shown in Figure 10. This design is based on the MT-Orbix thread filter framework, which implements a variant of the Chain of Responsibility pattern [31].

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. Common tasks include intercepting, modifying, or examining each request sent to and from the ORB.

In the thread framework architecture, a connection 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 the thread to handle this request. In the *thread-pool* model, the thread filter enqueues the request into a queue serviced by a thread with the appropriate priority. This thread then passes control back to the ORB, which performs operation demultiplexing and dispatches the upcall (**4**).

• **Advantages:** The main advantage of a threading framework is its flexibility. The thread filter mechanism can be programmed by server developers to support various concurrency strategies. For instance, to implement a thread-per-request [32] strategy, the filter can spawn a new thread and pass the request to this new thread. Likewise, the MT-Orbix threading framework can be configured to implement other concurrency architectures such as thread-per-object [33] and thread pool [34].

• **Disadvantages:** There are several disadvantages with the thread framework design, however. First, since there is only a single chain of filters, priority inversion can occur because 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 be processed only after several lower priority requests that arrived earlier. Third, the generality of the threading framework can substantially increase locking overhead since locks must be acquired to insert requests into the queue of the appropriate thread.

3.2.4 The Reactor-per-Thread-Priority Architecture

• **Overview:** The `Reactor`-per-thread-priority architecture is based on the Reactor pattern [35], which integrates transport endpoint demultiplexing and the dispatching of the corresponding event handlers. This threading architecture associates a group of `Reactors` with a group of threads running at different priorities. As shown in Figure 11, the compo-

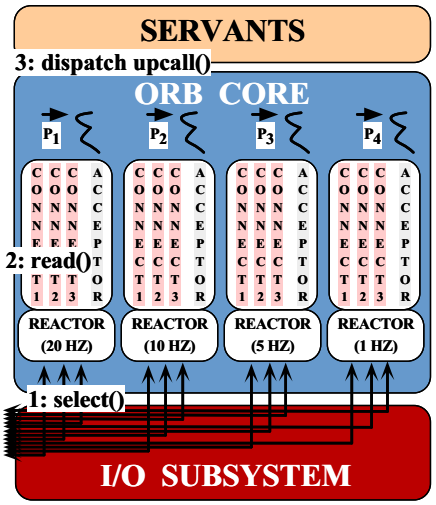

Figure 11: Server-side Reactor-per-Thread-Priority Concurrency Architecture

nents in the `Reactor`-per-thread-priority architecture include multiple pre-allocated `Reactors`, each of which is associated with its own real-time thread of control for each priority level, *e.g.*, $P_1 \ldots P_4$, in the ORB. For instance, avionics mission computing systems [36] 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.

Within each thread, the `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 executes the upcall at its thread priority.

Each `Reactor` in an ORB server thread is also associated with an `Acceptor` [37]. The `Acceptor` is a factory that listens on a particular port number for clients to connect to that thread and creates a connection handler to process the GIOP requests. In the example in Figure 11, there is a listener port for each priority level.

• **Advantages:** The advantage of the `Reactor`-per-thread-priority architecture is that it minimizes priority inversion and non-determinism. Moreover, it reduces context switching and synchronization overhead by requiring the state of servants to be locked only if they interact across different thread priorities. In addition, this concurrency architecture supports scheduling and analysis techniques that associate priority with rate, such as Rate Monotonic Scheduling (RMS) and Rate Monotonic Analysis (RMA) [38, 39].

• **Disadvantages:** The disadvantage with the `Reactor`-per-thread-priority architecture is that it serializes all client requests for each `Reactor` within a single thread of control, which can reduce parallelism. To alleviate this problem, a variant of this architecture can associate a *pool* of threads with each priority level. Though this will increase potential parallelism, it can incur greater context switching overhead and non-determinism, which may be unacceptable for certain types of real-time applications.

3.3 Integrating Connection and Concurrency Architectures

The `Reactor`-per-thread-priority architecture 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. In this diagram, the `Acceptors` listen on ports that

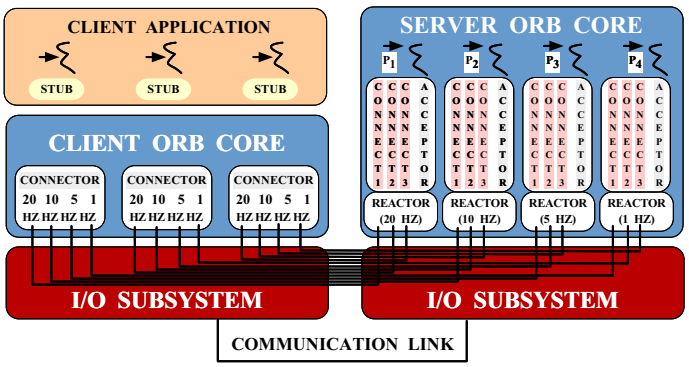

Figure 12: End-to-end Real-time ORB Core Software Architecture

correspond to the 20 Hz, 10 Hz, 5 Hz, and 1 Hz rate group thread priorities, respectively. Once a client connects, its `Acceptor` creates a new socket queue and connection handler to service that queue. The I/O subsystem uses the port number contained in arriving requests as a demultiplexing key to associate requests with the appropriate socket queue. Each queue is served by an ORB Core thread that runs at the appropriate priority.

The combination of the `Reactor`-per-thread-priority architecture and the non-multiplexed connection architecture minimizes priority inversion through the entire distributed ORB endsystem by eagerly demultiplexing incoming requests onto the appropriate real-time thread that services the priority

level of the target servant. As shown in Section 4.2, this design is well suited for real-time applications with deterministic QoS requirements.

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 MT-Orbix 2.2, Sun miniCOOL 4.3[2], 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. TAO has been used in a number of production real-time systems at companies like Boeing [36], SAIC, Lockheed Martin, and Raytheon.

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 used in the

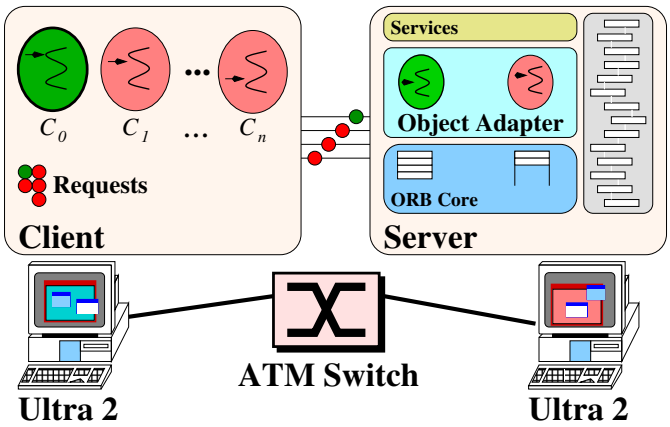

Figure 13: ORB Endsystem Benchmarking Testbed

experiments are outlined below.

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

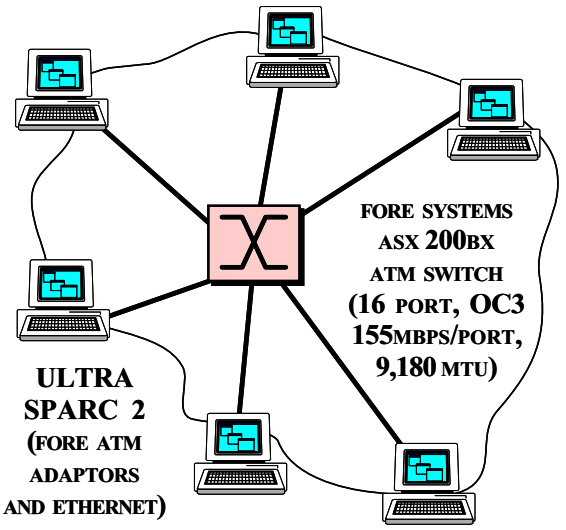

Figure 14: Hardware for the CORBA/ATM Testbed

4.1.1 Hardware Configuration

The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running Solaris 2.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSPARC-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The Solaris 2.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework [40].

Each UltraSPARC-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which 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 Kb). This allows up to eight switched virtual connections per card. The CORBA/ATM hardware platform is shown in Figure 14.

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 an ORB's 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 [41] 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 CORBAplus, multi-threaded applications have an event dispatcher thread and a pool of worker threads. The dispatcher thread receives the requests and passes them to application worker threads, which process the requests. In the simplest configuration, a server application can choose to create no additional threads and rely upon the main thread to process all requests.

• **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. Application developers can choose between thread-per-request or thread-pool. The thread-pool concurrency architecture was used for our benchmarks because it is better suited for deterministic real-time applications than thread-per-request. In the thread-pool concurrency architecture, the application initially spawns a fixed number of threads. In addition, when the initial thread pool size is insufficient, miniCOOL can be configured to handle requests on behalf of server applications by spawning threads dynamically up to a maximum limit.

• **MT-Orbix:** which uses the thread pool framework architecture based on the Chain of Responsibility pattern described in Section 3.2.3. Version 2.2 of MT-Orbix is used to create two real-time servant threads at startup. 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. This inversion could otherwise occur if a high-priority servant and a low-priority servant dequeued requests from the same queue.

• **TAO:** which uses the `Reactor`-per-thread-priority concurrency architecture described in Section 3.2.4. Version 1.0 of TAO integrates the `Reactor`-per-thread-priority concurrency architecture with a non-multiplexed connection architecture, as shown in Figure 15. In contrast, the other three ORBs multiplex all requests from client threads in each process over a single connection to the server process.

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 two-way calls per second. All low-priority clients have the same lower priority OS thread priority and invoke operations at 10 Hz, *i.e.*, they invoke 10 CORBA two-way 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.

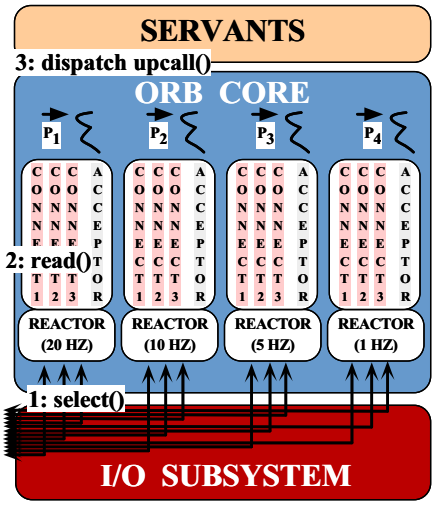

Figure 15: TAO's `Reactor`-per-Thread-Priority Thread Pool Architecture

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. These threads execute in an order determined by the Solaris real-time thread dispatcher. Each client invokes 4,000 CORBA two-way requests at its prescribed rate.

4.2 Performance Results on Solaris

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

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

Whitebox benchmarks: To precisely pinpoint the *sources* of priority inversion and performance non-determinism, we employed whitebox benchmarks. These benchmarks used profiling tools, such as UNIX `truss` and `Quantify` [42]. 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. In general, the results reveal why ORBs like MT-Orbix, CORBAplus, and miniCOOL are not yet suited for applications with real-time performance requirements. Likewise, the results illustrate empirically how and why the non-multiplexed, priority-based ORB Core architecture used by TAO is more suited for many types of real-time 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 exhibit no variance in the latency observed by the high-priority client, irrespective of the number of low-priority clients. Our measurements of end-to-end two-way ORB latency yielded the results in Figure 16.

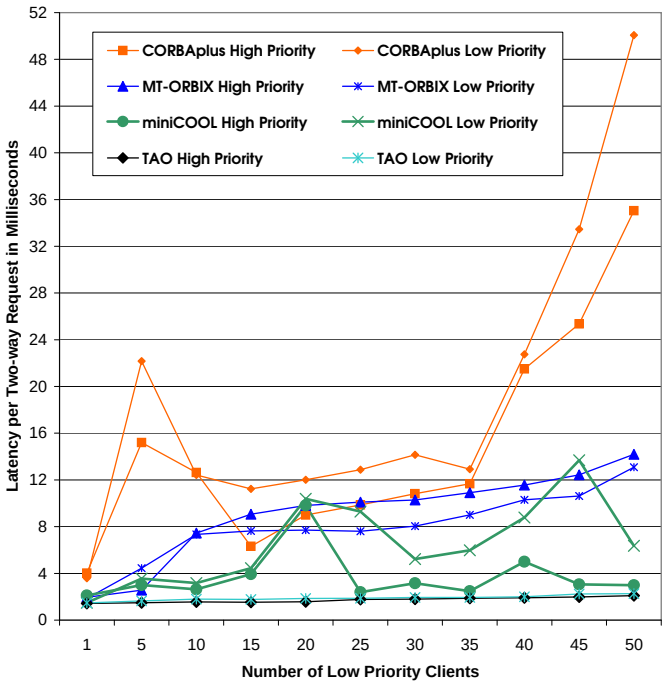

Figure 16: Comparative Latency for CORBAplus, MT-Orbix, miniCOOL, and TAO

Figure 16 shows that as the number of low-priority clients increases, MT-Orbix and CORBAplus incur significantly higher latencies for their high-priority client thread. Compared with TAO, MT-Orbix's latency is 7 times higher and CORBAplus' latency is 25 times higher. In addition, note the irregular behavior of the average latency that miniCOOL displays, *i.e.*, from 10 msec latency running 20 low-priority clients down to 2 msec with 25 low-priority clients. Such non-determinism is undesirable for real-time applications.

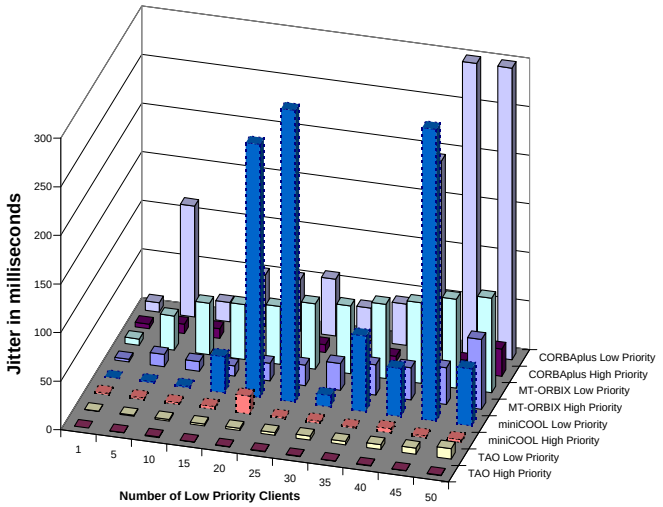

Figure 17: Comparative Jitter for CORBAplus, MT-Orbix, miniCOOL and TAO

The low-priority clients for MT-Orbix, CORBAplus and miniCOOL also exhibit very high levels of jitter. Compared with TAO, CORBAplus incurs 300 times as much jitter and MT-Orbix 25 times as much jitter in the worst case, as shown in Figure 17. Likewise, miniCOOL's low-priority clients display an erratic behavior with several high bursts of jitter, which makes it undesirable for deterministic real-time applications.

The blackbox results for each ORB are explained in more detail below.

CORBAplus results: CORBAplus incurs priority inversion at various points in the graph shown in Figure 16. After displaying a high amount of latency for a small number of low-priority clients, the latency drops suddenly at 10 clients, then eventually rises again. This erratic behavior is not suitable for deterministic real-time applications. Section 4.2.2 reveals how the poor performance and priority inversions stem largely from CORBAplus' concurrency architecture. Figure 17 shows that CORBAplus generates high levels of jitter, particularly when tested with 40, 45, and 50 low-priority clients. These results show an erratic and undesirable behavior for applications that require real-time guarantees.

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. This behavior is not conducive to deterministic real-time applications. Section 4.2.2 reveals how these inversions stem largely from the MT-Orbix's concurrency architecture on the server. In addition, MT-Orbix produces high levels of jitter, as shown in Figure 17. 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 also increases, reaching ∼10 msec, at 20 clients, at which point it decreases suddenly to 2.5 msec with 25 clients. This erratic behavior becomes more evident as more low-priority clients are run. Although the latency of the high-priority client is smaller than the low-priority clients, the non-linear behavior of the clients makes miniCOOL problematic for deterministic real-time applications.

The difference in latency between the high- and the low-priority client is also unpredictable. For instance, it ranges 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 17. This jitter is similar to that observed by the CORBAplus ORB since both spend approximately the same percentage of time executing locking operation. Section 4.2.2 evaluates ORB locking behavior.

TAO results: Figure 16 reveals that as the number of low-priority clients increase from 1 to 50, the latency observed by TAO's high-priority client grows by ∼0.7 msecs. However, the difference between the low-priority and high-priority clients starts at 0.05 msec and ends at 0.27 msec. In contrast, in miniCOOL, it grows from 0.55 msec to 10 msec, and in CORBAplus it grows from 0.42 msec to 15 msec. Moreover, the rate of increase of latency with TAO is significantly lower than MT-Orbix, Sun miniCOOL, and CORBAplus. In particular, when there are 50 low-priority clients competing for the CPU and network bandwidth, the low-priority client latency observed with MT-Orbix is more than 7 times that of TAO, the miniCOOL latency is ∼3 times that of TAO, and CORBAplus is ∼25 times that of TAO.

In contrast to the other ORBs, TAO's high-priority client always performs better than its low-priority clients. This demonstrates that the connection and concurrency architectures in TAO's ORB Core are better suited to maintaining real-time request priorities end-to-end. The key difference between TAO and other ORBs is that its 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 locking overhead.

The TAO ORB also exhibits very low jitter (less than 11 msecs) for the low-priority requests and lower 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 general, the blackbox results described above demonstrate that improper choice of ORB Core concurrency and connection software architectures can play a significant role in exacerbating priority inversion and non-determinism. The fact that TAO achieves such low levels of latency and jitter when run over the non-real-time Solaris I/O subsystem further demonstrates the feasibility of using standard OO middleware like CORBA to support real-time applications.

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 two-way requests to the server, which had a low-priority servant and high-priority servant thread.

Our previous experience using CORBA for real-time avionics mission computing [36] indicated 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 the ORBs consumed 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 (`mutex_lock` and `mutex_unlock`) and kernel-level locks (`_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 kernel/user mode switching overhead.

The whitebox results from our experiments are presented below.

CORBAplus whitebox results: Our whitebox analysis of CORBAplus reveals high levels of synchronization overhead from mutex and semaphore operations at the user-level for each two-way request, as shown in Figure 22. Synchronization overhead arises from locking operations that implement the connection and concurrency architecture used by CORBAplus.

As shown in Figure 18, CORBAplus exhibits high synchronization overhead (52%) using kernel-level locks in the client and the server incurs high levels of processing overhead (45%) due to kernel-level lock operations.

For each CORBA request/response, CORBAplus's client ORB performs 199 lock operations, whereas the server performs 216 user-level lock operations, as shown in Figure 22. This locking overhead stems largely from excessive dynamic memory allocation, as described in Section 4.4. 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.

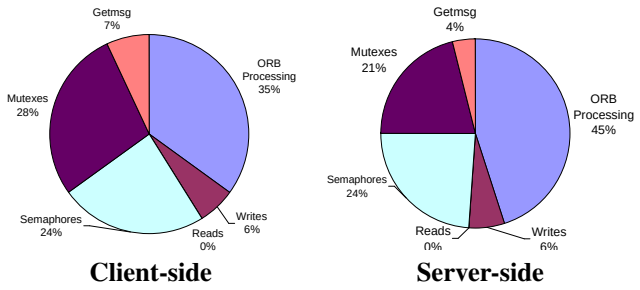

Figure 18: Whitebox Results for CORBAplus

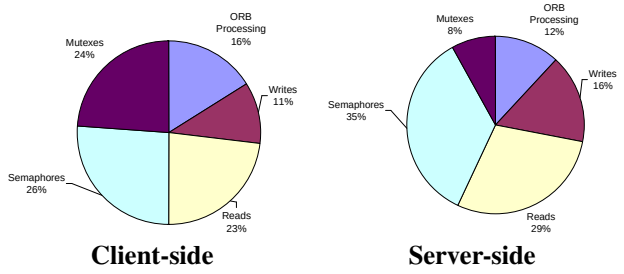

Figure 19: Whitebox Results for miniCOOL

• **CORBAplus connection architecture:** The CORBAplus ORB connection architecture uses the active connection model described in Section 3.1.1 and depicted in Figure 8. This design multiplexes all requests to the same server through one active connection thread, which simplifies ORB implementations by using a uniform queueing mechanism.

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

The CORBAplus connection architecture and the server concurrency architecture help reduce the number of simultaneous open connections and simplify the ORB implementation. However, concurrent requests to the shared connection incur high overhead because each send operation incurs a context switch. In addition, on the client-side, threads of different priorities can share the same transport connection, which can cause priority inversion. For instance, a high-priority thread may be blocked until a low-priority thread finishes sending its request. Likewise, the priority of the thread that blocks on the semaphore to receive a reply from a two-way 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 of miniCOOL reveals that synchronization overhead from mutex and semaphore operations consume a large percentage of the total miniCOOL ORB processing time. As with CORBAplus, synchronization overhead in miniCOOL arises from locking operations that implement its connection and concurrency architecture. Locking overhead accounted for ∼50% on the client-side and more than 40% on the server-side, as shown in Figure 19).

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 22. As with CORBAplus, this locking overhead stems largely from excessive dynamic memory allocation. Each dynamic allocation causes two user-level lock operations, *i.e.*, one acquire and one release.

The number of calls per-request to kernel-level locking mechanisms at the server (shown in Figure 23) are unusually high. This overhead stems from the fact that miniCOOL uses "system scoped" threads on Solaris, which require kernel intervention for all synchronization operations [43].

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 model described in Section 3.1.1. This architecture allows the leader thread to block in `select` on the shared socket. 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 `select`, thereby becoming the new leader.

• **miniCOOL concurrency architecture:** The Sun miniCOOL ORB concurrency architecture uses the leader/followers thread pool architecture described in Section 3.2.2. This architecture waits for connections in a single thread. 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. As with CORBAplus, 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 low-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 server-side of MT-Orbix.

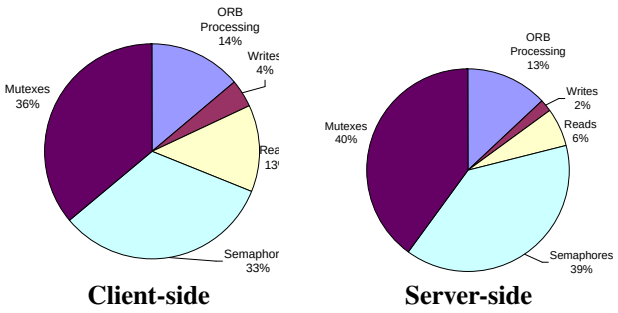

Figure 20: Whitebox Results for MT-Orbix

• **MT-Orbix connection architecture:** Like miniCOO MT-Orbix uses the leader/follower multiplexed connection a chitecture. Although this model minimizes context switchi overhead, it causes intensive priority inversions.

• **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.*, the high-priority servant had the highest 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 dispatch latency.

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. MT-Orbix provides only *one* filter chain. Thus, all incoming requests must be processed sequentially by the filters 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 the ORB processes it.

In addition, a filter can only be called after (1) GIOP processing has completed and (2) the Object Adapter has determined the target object for this request. This processing is serialized since the MT-Orbix ORB Core 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 MT-Orbix.

MT-Orbix's concurrency architecture is chiefly responsible for its substantial priority inversion shown in Figure 16. This figure shows how the latency observed by the high-priority client increases rapidly, growing from $\sim$2 msecs to $\sim$14 msecs as the number of low-priority clients increase from 1 to 50.

The MT-Orbix filter mechanism also 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 22. Moreover, MT-Orbix displays a high number of kernel-level locks per-request, as shown in Figure 23.

TAO whitebox results: As shown in Figure 21, TAO exhibits negligible synchronization overhead. TAO performs no

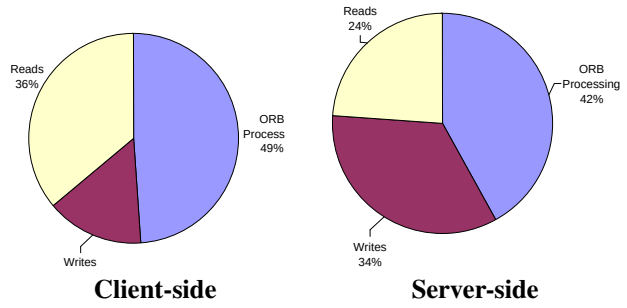

Figure 21: Whitebox Results for TAO

user-level lock operations on the client-side, nor on the server-side. This absence 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 uses no user-level locking operations and no kernel-level locks.

• **TAO connection architecture:** TAO uses a non-multiplexed connection architecture, which pre-establishes connections to servants, as described in Section 3.1.2. One connection is pre-established for each priority level, thereby avoiding the non-deterministic delay involved in dynamic connection setup. In addition, different priority levels have their own connection. This design avoids request-level priority inversion, which would otherwise occur from FIFO queueing *across* client threads with different priorities.

• **TAO concurrency architecture:** TAO supports several concurrency architectures, as described in [22]. The

`Reactor`-per-thread-priority architecture described in Section 3.2.4 was used for the benchmarks 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 than the high-priority client (10 Hz vs. 20 Hz, respectively).

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, *i.e.*, they have separate `Reactors`, `Acceptors`, Object Adapters, etc. In addition, the two threads service separate client connections, thereby eliminating the priority inversion that would 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 22) and kernel-level locking overhead (shown in Figure 23) 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 using global C++ operators `new` and `delete`. These operators allocate memory from a globally managed heap in each process.

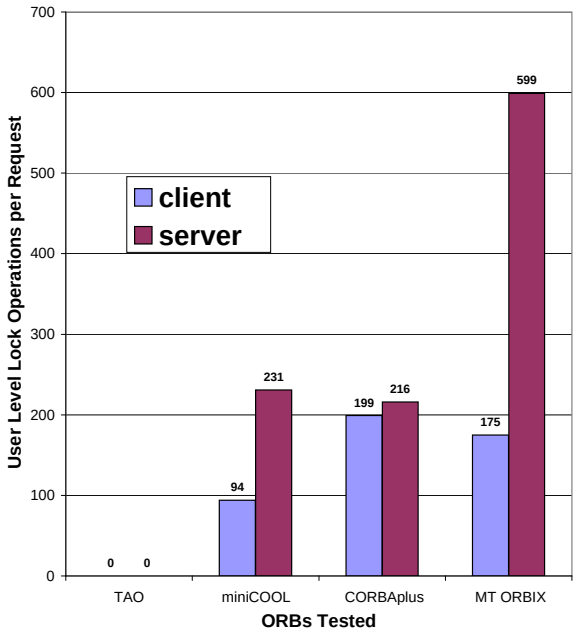

Figure 22: User-level Locking Overhead in ORBs

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 evaluated above arise from kernel-level lock operations.

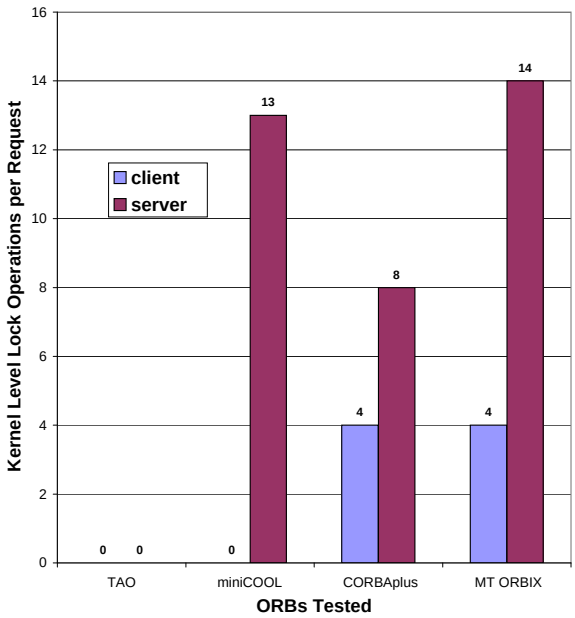

Figure 23: Kernel-level Locking Overhead in ORBs

Figures 22 and 23 illustrate how unlike the other three ORBs, TAO incurs no user-level or kernel-level locking. TAO can be configured so that ORB resources are not shared between its threads. For instance, it eliminates dynamic allocation, and the associated locking operations, by using buffers that are pre-allocated off the run-time stack. Each buffer is subdivided to accommodate the various fields of the request.

4.3 Performance Results on Chorus ClassiX

The performance results in Section 4.2 were obtained on Solaris 2.5.1, which provides real-time scheduling but not real-time I/O [41]. Therefore, Solaris cannot guarantee the availability of resources like I/O buffers and network bandwidth [22]. Moreover, the scheduling performed by the Solaris I/O subsystem is not integrated with the rest of its resource management strategies.

So-called real-time operating systems typically provide mechanisms for priority-controlled access to OS resources. This allows applications to ensure that QoS requirements are met. QoS mechanisms provided by real-time operating systems typically include real-time scheduling classes that enforce QoS usage policies, as well as real-time I/O to specify

processing requirements and operation periods.

Chorus[3] ClassiX is a real-time OS that can scale down to small embedded configurations, as well as scale up to distributed POSIX-compliant platforms [44]. ClassiX provides a real-time scheduler that supports several scheduling algorithms, including priority-based FIFO preemptive scheduling. It supports real-time applications and general-purpose applications.

The IPC mechanism used on ClassiX, Chorus IPC, provides an efficient, location-transparent message-based communication facility on a single board and between multiple interconnected boards. In addition, ClassiX has a TCP/IP protocol stack, accessible via the Socket API, that enables internetworking connectivity with other OS platforms.

To determine the impact of a real-time OS on ORB performance, this subsection presents blackbox results for TAO and miniCOOL using ClassiX.

4.3.1 Hardware Configuration:

The following experiments were conducted using two MVME177 VMEbus single-board computers. Each MVME177 contains a 60 MHz MC68060 processor and 64 Mbytes of RAM. The MVME177 boards are mounted on a MVME954A 6-slot, 32-bit, VME-compatible backplane. In addition, each MVME177 module has an 82596CA Ethernet transceiver interface.

4.3.2 Software Configuration:

The experiments were run on version 3.1 of ClassiX. The ORBs benchmarked were miniCOOL 4.3 and TAO 1.0. The client/server configurations run were (1) locally, *i.e.*, client and server on one board and (2) remotely, *i.e.*, between two MVME177 boards on the same backplane.

The client/server benchmarking configuration implemented is the same[4] as the one run on Solaris 2.5.1 that is described in Section 4.1.2. MiniCOOL was configured to send messages on one board or across boards using the Chorus IPC communication facility, which is more efficient than the Chorus TCP/IP protocol stack. In addition, we conducted benchmarks of miniCOOL and TAO using the TCP protocol. In general, miniCOOL performs more predictably using Chorus IPC as its transport mechanism.

4.3.3 Blackbox results:

We computed the average two-way response time incurred by various clients. In addition, we computed two-way operation jitter. High levels of latency and jitter are undesirable for real-time applications since they complicate the computation of worst-case execution time and reduce CPU utilization.

miniCOOL using Chorus IPC: As the number of low-priority clients increase, the latency observed by the remote high- and low-priority client also increases. It reaches ∼34 msec, increasing linearly, when the client and the server are on different processor boards (remote) as shown in Figure 24.

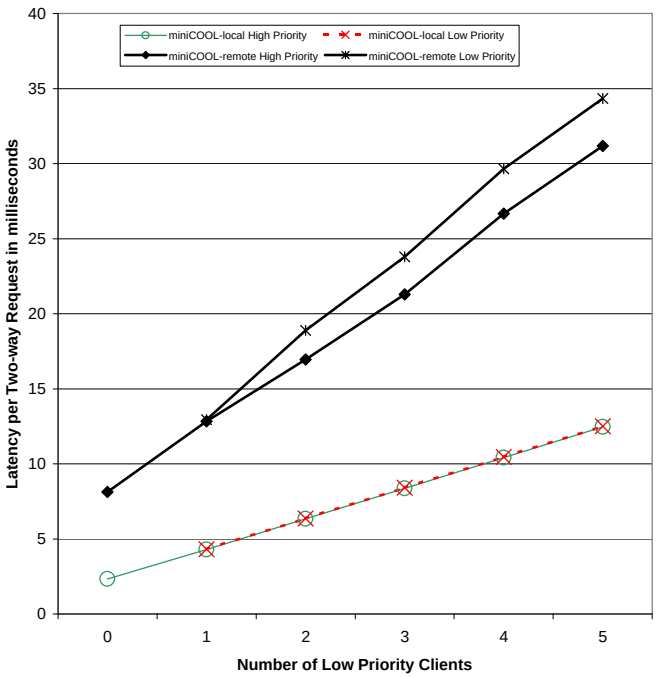

Figure 24: Latency for miniCOOL with Chorus IPC on ClassiX

When the client and server are collocated, the behavior is more stable on both the high and low-priority client, *i.e.*, they are essentially identical since their lines in Figure 24 overlap. The latencies start at ∼2.5 msec of latency and reaches ∼12.5 msecs. Both high- and low-priority clients incur approximately the same average latency.

In all cases, the latency for the high-priority client is always lower than the latency for the low-priority client. Thus, there is no significant priority inversion, which is expected for a real-time system. However, there is still variance in the latency observed by the high-priority client, in both, the remote and local configurations.

In general, miniCOOL performs more predictably on ClassiX than its version for Solaris. This is due to the use of TCP on Solaris versus Chorus IPC on ClassiX. The Solaris latency and jitter results were relatively erratic, as shown in the blackbox results from Solaris described in Section 4.2.1.

[3] Chorus has been purchased by Sun Microsystems.

[4] Note the number of low-priority clients used was 5 rather than 50 due to a bug in ClassiX that caused `select` to fail if used to wait for events on more than 16 sockets.

Figure 25 shows that as the number of low-priority clients increases, the jitter increases progressively manner, for remote high- and low-priority clients. In addition, Figure 25 illustrates

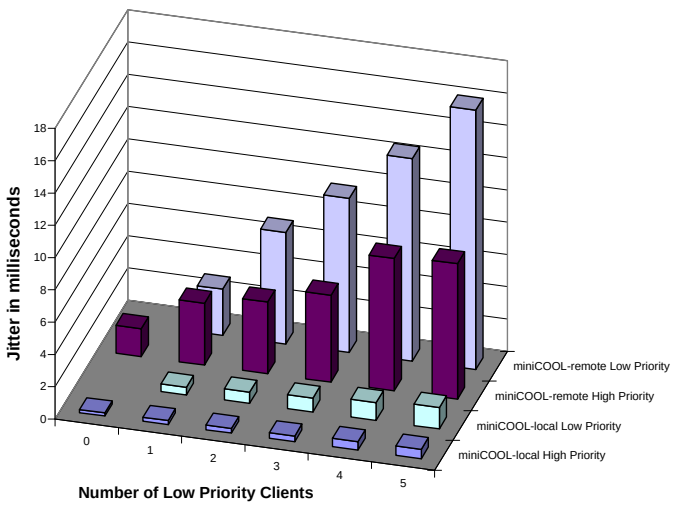

Figure 25: Jitter for miniCOOL with Chorus IPC on ClassiX

that the jitter incurred by miniCOOL's remote clients is fairly high. The unpredictable behavior of high- and low-priority clients is more evident when the client and the server run on separate processor boards, as shown in Figure 24. Moreover, Figure 24 illustrates the difference in latency between the local and remote configurations, which appears to stem from the latency incurred by the network I/O driver.

miniCOOL using TCP: We also configured the miniCOOL client/server benchmark to use the Chorus TCP/IP protocol stack. The TCP/IP implementation on ClassiX is not as efficient as Chorus IPC. However, it provided a base for comparison between miniCOOL and TAO, which uses TCP as its transport protocol.

The results we obtained for miniCOOL over TCP show that as the number of low-priority clients increase, the latency observed by the remote high- and low-priority client also increased linearly. The maximum latency was ∼59 msec, when the client and the server are on the same processor board (local) as shown in Figure 26.

The increase in latency for the local configuration is unusual since one would expect the ORB to perform best when client and server are collocated on the same processor. However, when client and server reside in different processor boards, illustrated in Figure 27, the average latency was more stable. This appears to be due to the implementation of the TCP/IP protocol stack, which may not to be optimized for local IPC.

When the client and server are on separate boards, the behavior is similar to the remote clients using Chorus IPC. This indicates that some bottlenecks reside in the Ethernet driver.

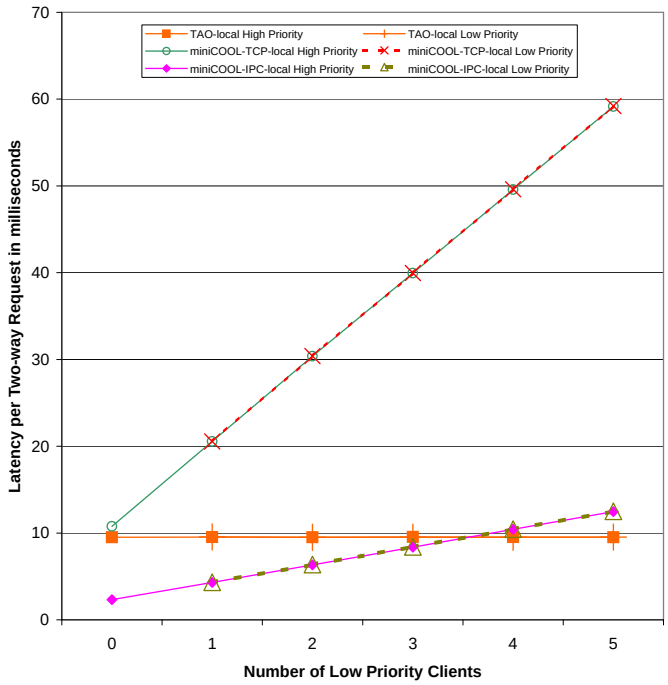

Figure 26: Latency for miniCOOL-TCP, miniCOOL-IPC, and TAO-TCP on ClassiX, local configuration

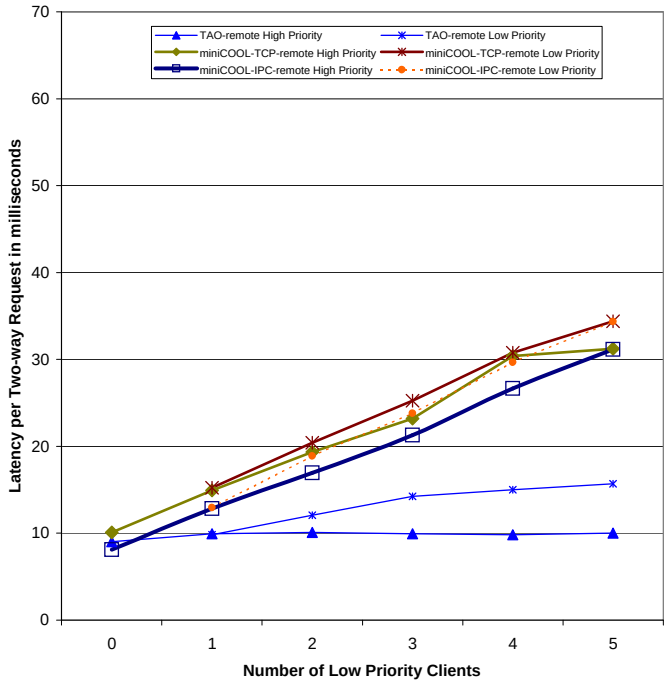

Figure 27: Latency for miniCOOL-TCP, miniCOOL-IPC, and TAO-TCP on ClassiX, remote configuration

In all cases, the latency for the high-priority client is always lower than the latency for the low-priority client, *i.e.*, there appears to be no significant priority inversion, which is expected for a real-time system. However, there is still variance in the latency observed by the high-priority client, in both the remote and local configurations, as shown in Figure 28. The remote configurations incurred the highest variance, with the exception of TAO's remote high-priority clients, whose jitter remained fairly stable. This stability stems from TAO's `Reactor`-per-thread-priority concurrency architecture described in Section 3.2.4.

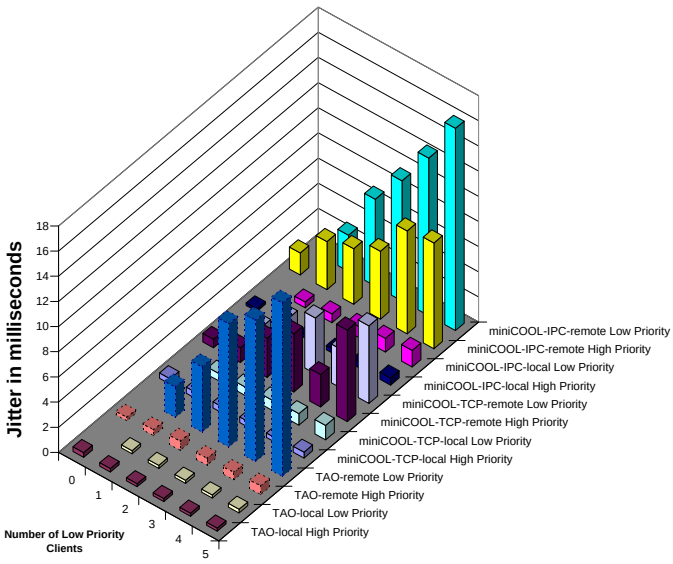

Figure 28: Jitter for miniCOOL-TCP, miniCOOL-IPC and TAO-TCP on ClassiX

TAO using TCP: Figure 26 reveals that as the number of low-priority clients increase from 0 to 5, the latency observed by TAO's high-priority client grows by ∼0.005 msecs for the local configuration and Figure 27 shows ∼1.022 msecs for the remote one. Although the remote high-priority client performs as well as the local one, the difference between the low-priority and high-priority remote clients evolves from 0 msec to 6 msec. This increase is unusual and appears to stem from factors external to the ORB, such as the ClassiX OS scheduling algorithm and network latency. In general, TAO performs more predictably in other platforms tested with higher bandwidth, *e.g.* 155 Mbps ATM networks. The local client/server test, in contrast, perform very predictably and have little increase in latency.

The TAO ORB produces very low jitter, less than 2 msecs, for the low-priority requests and lower jitter (less than 1 msec) for the high-priority requests. On this platform, the exception is the remote low-priority client, which may be attributed to the starvation of the low-priority clients by the high-priority one, and the latency incurred by the network. The stability of TAO's latency is clearly desirable for applications that require predictable end-to-end performance.

4.4 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 inversion and non-determinism are significant problems in conventional ORBs. As a result, these ORBs are not currently suitable for applications with deterministic real-time requirements. Based on our results, and our prior experience [14, 15, 16, 17] 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; TAO provides these APIs as extensions to CORBA.

We recommend that ORBs be enhanced to allow pre-establishment of connections in accordance to the "explicit binding" mechanism provided in the forthcoming OMG standard for real-time CORBA [45, 7].

2. Real-time ORBs should minimize dynamic memory management: 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, which releases allocated memory, also requires user-level locking to update the global shared pool. This lock sharing contributes to the overhead shown in Figure 22. In addition, locking also increases non-determinism due to contention and queueing.

We recommend that real-time ORBs avoid excessive sharing of dynamic memory locks via the use of mechanisms such as thread-specific storage [29], which allocates memory from separate heaps that are unique to each thread.

3. Real-time ORBs should avoid multiplexing requests of different priorities over a shared connection: Sharing connections among multiple threads requires synchronization. Not only does this increase locking overhead, but it also increases opportunities for priority inversion. For instance, high-priority requests can be blocked until low-priority threads release the shared connection lock. Priority inversion can be further exacerbated if multiple threads with multiple levels of thread priorities share common locks. For instance, medium

priority threads can preempt a low-priority thread that is holding a lock required by a high-priority thread, which can lead to unbounded priority inversion [13].

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

4. Real-time ORB concurrency architectures should be flexible, efficient, and predictable: Many ORBs, such as miniCOOL and CORBAplus, create threads on behalf of server applications. This design is inflexible because it prevents application developers from customizing ORB performance via a different concurrency architecture. Conversely, other ORB concurrency architectures are flexible, but inefficient and unpredictable, as shown by Section 4.2.2's 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 `Reactor`-per-thread-priority, thread pool, and thread-per-connection. Developers can configure TAO [46] to minimize unnecessary sharing of ORB resources by using thread-specific storage.

5. Real-time ORBs should avoid reimplementing OS mechanisms: Conventional ORBs incur substantial performance overhead because they reimplement native OS mechanisms for endpoint demultiplexing, queueing, and concurrency control. For instance, much of the priority inversion and nondeterminism miniCOOL, CORBAplus, and MT-Orbix stem from the complexity of their ORB Core mechanisms for multiplexing multiple client threads through a single connection to a server. These mechanism reimplement the connection management and demultiplexing features in the OS in a manner that (1) increases overhead and (2) does not consider the priority of the threads that make the requests for two-way operations.

We recommend that real-time ORB developers attempt to use the native OS mechanisms as much as possible, *e.g.*, designing the ORB Core to work in concert with the underlying mechanisms rather than reimplementing them at a higher level. A major reason that TAO performs predictably and efficiently is because the connection management and concurrency model used in its ORB Core is closely integrated with the underlying OS features.

6. The design of real-time ORB endsystem architectures should be guided by empirical performance benchmarks: Our prior research on pinpointing performance bottlenecks and optimizing middleware like Web servers [47, 48] and CORBA ORBs [15, 14, 17, 16] demonstrates the efficacy of a measurement-driven research methodology.

We recommend that ORB vendors and end-users work with the OMG to standardize real-time CORBA benchmarking techniques and metrics [49]. These benchmarks will simplify communication between researchers and developers. In addition, they will facilitate the comparison of performance results and real-time ORB behavior patterns between different ORBs and different OS/hardware platforms. The real-time ORB benchmarking test suite described in this section is available at `www.cs.wustl.edu/~schmidt/TAO.html`.

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 [50]. They identified key requirements and outlined mechanisms for supporting end-to-end timing constraints [51]. 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 [52]. 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 [51]. 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) [53]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed operation invocations (`TDMIs`) [54]. A `TDMI` corresponds to TAO's `RT_Operation` [22] 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` [51] express required timing constraints, *e.g.*, deadlines relative to the current time. In contrast, TAO's `RT_Operations` publish their resource requirements, *e.g.*, CPU time. 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 [55] has defined a model for communicating changes in QoS characteristics between applications, middleware, and the underlying endsystems and net-

work. 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) [56]. Though we believe 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 [57] 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 empirical results are not yet available.

The ARMADA project [58] 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 suitable for distributed, real-time applications with deterministic QoS requirements. Meeting these demands requires that ORB Core software architectures be designed to reduce priority inversion and increase end-to-end determinism.

The TAO ORB Core described in this paper reduces priority inversion and enhances determinism by using a priority-based concurrency architecture and non-multiplexed connection architecture that share a minimal amount of resources among threads within a process. The architectural principles used in TAO can be applied to other ORBs and other real-time software systems. Furthermore, our results demonstrate the feasibility of using OO middleware like CORBA to develop real-time applications that can perform well over (1) standard Internet protocols, (2) upper layer protocols such as IIOP that are based on the Internet protocols, and (3) off-the-shelf hardware/software.

TAO has been used to develop a number of real-time applications, including a real-time audio/video streaming service [59] and a real-time ORB endsystem for avionics mission computing [36]. The avionics application manages sensors and operator displays, navigates the aircraft's course, and controls weapon release. To meet the scheduling demands of real-time applications, TAO supports predictable scheduling and dispatching of periodic processing operations [10], as well as efficient event filtering and correlation mechanisms [36]. The C++ source code for TAO and ACE is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

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 and Bil Lewis for their 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*, 2.2 ed., Feb. 1998.

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

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

[8] Object Management Group, *Minimum CORBA - Request for Proposal*, OMG Document orbos/97-06-14 ed., June 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] A. Campbell and K. Nahrstedt, *Building QoS into Distributed Systems*. London: Chapman & Hall, 1997. Proceedings of the IFIP TC6 WG6.1 Fifth International Workshop on Quality of Service (IWQOS '97), 21-23 May 1997, New York.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Z. D. Dittia, J. R. Cox, Jr., 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.

[19] 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.

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

[21] 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*, Sept. 1999.

[22] 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), IEEE, June 1999.

[23] J. C. Mogul and A. Borg, "The Effects of Context Switches on Cache Performance," in *Proceedings of the* 4^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Santa Clara, CA), ACM, Apr. 1991.

[24] D. L. Levine, S. Flores-Gaitan, and D. C. Schmidt, "Measuring OS Support for Real-time CORBA ORBs," in *Proceedings of the* 4^{th} *Workshop on Object-oriented Real-time Dependable Systems*, (Santa Barbara, CA), IEEE, January 1999.

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

[26] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[27] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

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

[29] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently," *C++ Report*, vol. 9, November/December 1997.

[30] 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.

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

[32] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request," *C++ Report*, vol. 8, February 1996.

[33] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object," *C++ Report*, vol. 8, July 1996.

[34] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool," *C++ Report*, vol. 8, April 1996.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

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

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

[43] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[44] M. Guillemont, "CHORUS/ClassiX r3 Technical Overview (technical report #CS/TR-96-119.13)," tech. rep., Chorus Systems, May 1997.

[45] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

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

[47] 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.

[48] 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.

[49] S. Nimmagadda, C. Liyanaarchchi, D. Niehaus, A. Gopinath, and A. Kaushal, "Performance Patterns: Automated Scenario Based ORB Performance Evaluation," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[50] 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.

[51] G. Cooper, L. C. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thuraisingham, 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.

[52] "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.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] Object Management Group, *Control and Management of Audio/Video Streams: OMG RFP Submission*, 1.2 ed., Mar. 1997.

[60] 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, programming language, OS platform, communication protocols and interconnects, and hardware. Figure 29 illustrates the key components in the CORBA reference model that collaborate to provide this degree of portability, interoperability, and transparency.[5] Each component in the CORBA reference model is outlined below:

Client: A client is a role that obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, *i.e.*, `object→operation(args)`. Figure 29 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

[5]This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [5].

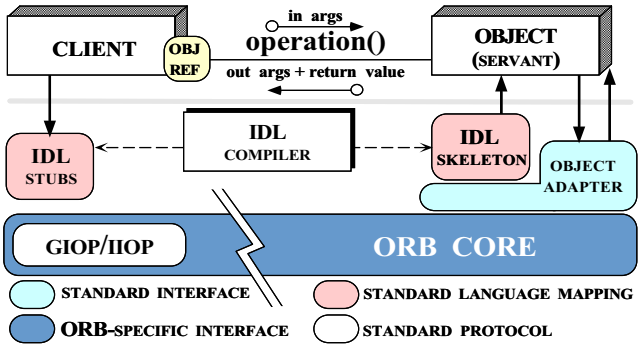

Figure 29: Key Components in the CORBA 2.x Reference Model

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

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

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

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [31] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common data-level representation. Conversely, skeletons implement the *Adapter* pattern [31] and demarshal the data-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language like C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [60].

Object Adapter: An Object Adapter associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. CORBA 2.2 portability enhancements [5] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.