Supporting High-performance I/O in QoS-enabled ORB Middleware

Fred Kuhns and David Levine
{fredk,levine}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130

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

This paper will appear in *Cluster Computing: the Journal on Networks, Software, and Applications*.

Abstract

To be an effective platform for high-performance distributed applications, off-the-shelf Object Request Broker (ORB) middleware, such as CORBA, must preserve communication-layer quality of service (QoS) properties both vertically (i.e., network interface ↔ application layer) and horizontally (i.e., end-to-end). However, conventional network interfaces, I/O subsystems, and middleware interoperability protocols are not well-suited for applications that possess stringent throughput, latency, and jitter requirements. It is essential, therefore, to develop vertically and horizontally integrated ORB endsystems that can be (1) configured flexibly to support high-performance network interfaces and I/O subsystems and (2) used transparently by performance-sensitive applications.

This paper provides three contributions to research on high-performance I/O support for QoS-enabled ORB middleware. First, we outline the key research challenges faced by high-performance ORB endsystem developers. Second, we describe how our real-time I/O (RIO) subsystem and pluggable protocol framework enables ORB endsystems to preserve high-performance network interface QoS up to applications running on off-the-shelf hardware and software. Third, we illustrate empirically how highly optimized ORB middleware can be integrated with real-time I/O subsystem to reduce latency bounds on communication between high-priority clients without unduly penalizing low-priority and best-effort clients. Our results demonstrate how it is possible to develop ORB endsystems that are both highly flexible and highly efficient.

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

1 Introduction

1.1 Current Limitations of High-performance Distributed Computing

During the past decade, there has been substantial R&D emphasis on *high-performance networking* and *performance optimizations* for network elements and protocols. This effort has paid off such that networking products are now available off-the-shelf that can support Gbps on every port, *e.g.*, Gigabit Ethernet and ATM switches. Moreover, 622 Mbps ATM connectivity in WAN backbones are becoming standard and 2.4 Gbps is starting to appear. In networks and GigaPoPs being deployed for the Next Generation Internet (NGI), such as the Advanced Technology Demonstration Network (ATDnet), 2.4 Gbps (OC-48) link speeds have become standard. However, the general lack of flexible software tools and standards for programming, provisioning, and controlling these networks has limited the rate at which applications have been developed to leverage advances in high-performance networks.

During the same time period, there has also been substantial R&D emphasis on object-oriented (OO) communication *middleware*, including open standards like OMG's Common Object Request Broker Architecture (CORBA) [1], as well as popular proprietary solutions like Microsoft's Distributed Component Object Model (DCOM) [2] and Sun's Remote Method Invocation (RMI) [3]. These efforts have paid off such that OO middleware is now available off-the-shelf that allows clients to invoke operations on distributed components without concern for component location, programming language, OS platform, communication protocols and interconnects, or hardware [4].

However, off-the-shelf communication middleware has several limitations. In particular, it historically has lacked (1) support for QoS specification and enforcement, (2) integration with high-performance networking technology, and (3) performance, predictability, and scalability optimizations [5]. These omissions have limited the rate at which performance-sensitive applications, such as video-on-demand, teleconferencing, and scientific computing, have been developed to leverage advances in communication middleware.

1.2 Overcoming Current Limitations with TAO

To address the flexibility and QoS performance limitations outlined above, we have developed *The ACE ORB* (TAO) [5]. TAO is a high-performance, real-time Object Request Broker (ORB) endsystem targeted for applications with deterministic and statistical QoS requirements, as well as best effort requirements. The TAO ORB endsystem contains the network interface, OS I/O subsystem, communication protocol, and CORBA-compliant middleware components and features shown in Figure 1.

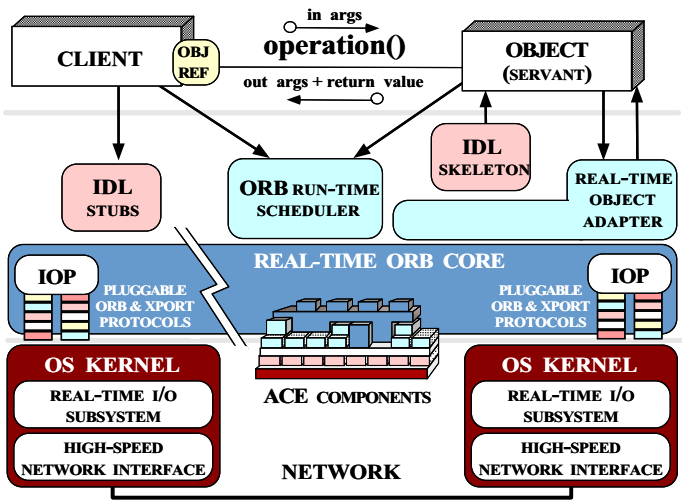

Figure 1: Components in the TAO ORB Endsystem

TAO's ORB Core, Object Adapter, stubs/skeletons, and servants run in user-space and handle connection management, data transfer, endpoint and request demultiplexing, concurrency, (de)marshaling, and application operation processing. In contrast, TAO's real-time I/O (RIO) subsystem runs in the OS kernel and sends/receives requests to/from clients across high-performance networks or I/O backplanes. TAO's pluggable protocol framework provides the "glue" that integrates its higher-level ORB Core and Object Adapter components with its lower-level I/O subsystem and network interface components. Developers can use TAO's pluggable protocols framework to implement new ORB messaging protocols and transport adapters that leverage underlying high-performance communication protocols and network interface hardware.

1.3 Key Research Challenges

This paper focuses on the techniques used in TAO to resolve the following research challenges that arise when developing high-performance and real-time ORB endsystems:

1. Optimizing QoS-enabled I/O subsystems to support high-performance network interfaces: A key ORB endsystem research challenge is to implement and optimize QoS-enabled OS I/O subsystems and network interfaces. This paper presents the design and performance of a real-time I/O (RIO) subsystem that enhances the Solaris 2.5.1 kernel to enforce the QoS features in TAO's ORB endsystem [6]. RIO provides QoS guarantees for vertically integrated ORB endsystems that can (1) increase throughput, (2) decrease latency, and (3) improve end-to-end predictability for distributed applications. RIO supports periodic protocol processing, guarantees I/O resources to applications, and minimizes the effect of flow control in communication streams end-to-end. A novel feature of the RIO subsystem is its integration of real-time scheduling and protocol processing, which allows RIO to support both bandwidth guarantees and low-delay applications.

Many RIO mechanisms and features are implemented in its network interface drivers, which makes it uniquely suited for high-performance network interface technology. For example, network interface architectures, such as the ATM Port Interconnect Controller (APIC) [7], U-Net [8], and the Virtual Interface Architecture (VIA [9]), support the vertical integration of I/O subsystems, which can minimize extraneous memory reads and writes using advanced I/O techniques, such as protected DMA [10]. RIO can exploit these techniques to support user-space protocol implementations and zero-copy I/O. Likewise, network interface drivers can take advantage of RIO to support intelligent polling, periodic I/O, co-scheduling of user- and kernel-threads, and strategized buffer management.

2. Developing a pluggable protocols framework to integrate new protocols that can leverage high-performance network interfaces: Another key ORB endsystem research challenge is to integrate the optimized, QoS-enabled I/O subsystem and network interfaces with higher-level OO middleware features, such as parameter (de)marshaling, request demultiplexing, concurrency control, and fault tolerance. This requires an efficient framework supporting custom protocols that leverage underlying, platform-specific hardware/OS features. However, the framework must also meet the seemingly contradictory goal of providing a flexible platform-neutral API to the ORB and applications.

TAO's pluggable protocols framework can be used to create custom inter-ORB protocols or to exploit features of the specialized hardware or software within the OS I/O subsystem. For example, if a VIA interface is used a specialized transport adaptor can be developed that is optimized for VIA features. In particular, this adaptor can leverage the shared memory model offered by VIA and integrate it with the ORB's internal request buffering mechanisms, thereby yielding more efficient memory management.

1.4 Paper Organization

The paper is organized as follows: Section 2 provides a general overview of the TAO ORB endsystem architecture; Section 3 describes how the RIO subsystem enhances the Solaris 2.5.1 OS kernel to support end-to-end QoS for TAO applications; Section 4 describes TAO's pluggable protocols framework; Section 5 illustrates how TAO's RIO subsystem and pluggable protocols framework can seamlessly leverage high-performance network interfaces; Section 6 presents empirical results from systematically benchmarking the efficiency and predictability of TAO and RIO over an ATM network; Section 7 compares RIO and TAO's pluggable protocols framework with related work; Section 8 presents concluding remarks. For completeness, Appendix A provides a brief overview of CORBA and Appendix B provides and overview of the Solaris operating system.

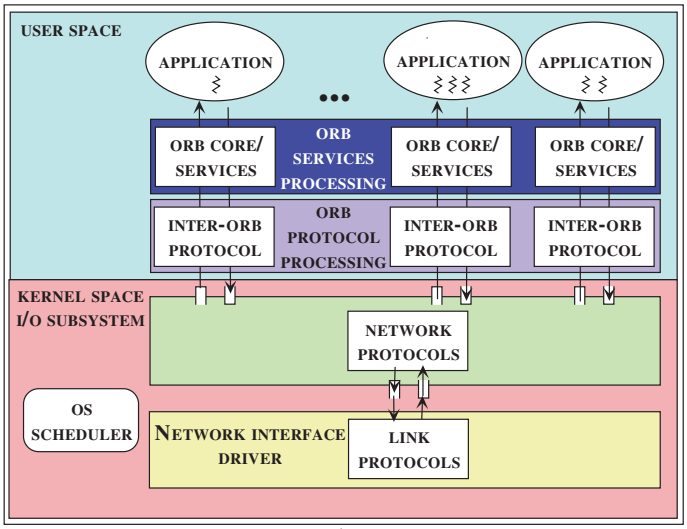

Figure 2: Typical ORB Endsystem Architecture

2 Supporting High-performance I/O in the TAO Endsystem

As outlined in Section 1, key research challenges faced when integrating high-performance I/O within an ORB endsystem involve optimizing I/O subsystem mechanisms and ORB inter-ORB protocols to exploit the underlying hardware and I/O subsystem. In this section we outline an ORB endsystem architecture that addresses these two challenges.

2.1 Context

As shown in Figure 2, CORBA endsystems can be divided into several components, including the operating system's I/O subsystem, Inter-ORB protocol processing, and ORB Core services. In this environment application threads acquire object references, invoke remote operations, and performance application-specific processing. CORBA objects are implemented within the context of a server ORB, which is responsible for performing incoming upcalls on target objects and sending replies back to the clients.[1] Moreover, CORBA supports both synchronous and asynchronous invocation models [11].

2.2 Design Challenges

Developers of ORB endsystem that use general-purpose operating systems, such as Solaris, Windows NT, or NetBSD, must address the following design challenges in order to meet the QoS requirements of high-performance and real-time applications.

[1] CORBA client applications also can issue one-way invocations where no reply is sent from the server.

2.2.1 Alleviating Thread-based Priority Inversions

Thread-based priority inversion can occur when a higher priority thread blocks awaiting a resource held by a lower priority thread [12]. When considering the kernel or I/O subsystem, this type of priority inversion occurs when real-time application threads depend on system level processing which is performed at priority levels that are either higher or lower than the currently running thread. For example, the Solaris kernel performs protocol processing asynchronously at either SYS or INT priorities [13, 14, 12].[2] Likewise, in NetBSD the processing of the callout queue that service TCP timeouts occurs at a software interrupt priority that preempts all application-level threads.

As illustrated above, thread-based priority inversion occur when either the kernel performs asynchronous or synchronous processing without regard to the relative priority of the benefiting thread or process. In practice this generally translates to one of two cases: 1) system processing performed by kernel threads with their own scheduling attributes or 2) event processing with hardware or software interrupt priorities.

Asynchronous processing with kernel threads: Modern UNIX operating systems, such as Solaris, rely on kernel threads to perform asynchronous or synchronous system activities, such as callout queue processing, page replacement, or processing STREAMS `svc` functions. In Solaris, these threads operate with global priorities that are lower than those for the real-time scheduling class. Other operating systems perform similar processing using software interrupt priorities. In both cases, application threads can experience unbounded priority inversion [15].

[2] Appendix B presents an overview of the Solaris scheduling model.

Protocol processing with interrupt priorities: Another source of thread-based priority inversion occurs when protocol processing of incoming packets is performed in an interrupt context. Traditional UNIX implementations treat all incoming packets with equal priority, regardless of the priority of the application thread that ultimately receives the data.

In BSD UNIX-based systems [16], for instance, the interrupt handler for the network driver deposits the incoming packet in the IP queue and schedules a software interrupt that invokes the `ip_input` function. Before control returns to the interrupted application process, the software interrupt handler is run and `ip_input` is executed. The `ip_input` function executes at the lowest interrupt level and processes all packets in its input queue. Only when this processing is complete does control return to the interrupted process. Thus, not only is the process preempted, but it will be charged for the CPU time consumed by input protocol processing.

In STREAMS-based UNIX operating systems, protocol processing can either be performed in an interrupt context (as in Solaris) or with `svc` functions scheduled asynchronously. Using `svc` functions can yield the unbounded priority inversion described above. Similarly, processing all input packets in an interrupt context can cause unbounded priority inversion.

Modern high-speed network interfaces can saturate the system bus, memory, and CPU, leaving little time available for application processing. It has been shown [14] that if protocol processing on incoming data is performed in an interrupt context this can lead to *input livelock*. Livelock is a condition where the overall endsystem performance degrades due to input processing of packets in an interrupt context. In extreme cases, an endsystem can spend the majority of its time processing input packets, resulting in little or no useful work being done. Thus, input livelock can prevent an ORB endsystem from meeting its QoS commitments to applications.

2.2.2 Alleviating Packet-based Priority Inversions

Packet-based priority inversion [15], also known as "head-of-line" blocking, can occur when packets for high-priority applications are queued behind packets for low-priority application threads. This inversion can occur as a result of serializing the processing of incoming or outgoing network packets. To meet deadlines of time-critical applications, it is important to eliminate, or at least minimize, packet-based priority inversion.

To illustrate this problem, consider a general-purpose ORB endsystem that must support both soft real-time applications, such as audio/video (A/V) conferencing [17], and "best-effort" applications, such as remote file transfer. This endsystem must transmit both (1) time critical video frames and audio packets and (2) relatively low-priority file buffers. For the system to operate correctly, A/V frames must be delivered periodically with strict bounds on latency and jitter. Conversely, bulk data file transfers occur periodically and inject a large number of packets into the I/O subsystem, which are queued at the network interface. Unfortunately, packets containing high-priority A/V frames can be queued in the network interface *behind* low-priority bulk data packets containing file buffers, thereby yielding packet-based priority inversion. Thus, A/V frames may arrive too late to meet end-to-end application QoS requirements.

2.2.3 Alleviating Limitations with Inter-ORB Protocol Implementations

CORBA's standard interoperability protocols are well-suited for conventional request/response applications with best-effort QoS requirements [18]. They are not well-suited, however, for high-performance real-time and/or embedded applications that cannot tolerate the message footprint size or the latency, overhead, and jitter of the TCP/IP-based Inter-ORB transport protocol [19]. For instance, TCP functionality, such as adaptive retransmissions, deferred transmissions, and delayed acknowledgments, can cause excessive overhead and latency for real-time applications [20]. Likewise, best-effort networking protocols, such as IPv4, lack the functionality of packet admission policies and rate control, which can lead to excessive congestion and missed deadlines in networks and endsystems.

Therefore, applications with stringent QoS requirements need optimized protocol implementations, QoS-aware interfaces, custom presentations layers, specialized memory management (*e.g.*, shared memory between ORB and I/O subsystem), and alternative transport programming APIs (*e.g.*, sockets vs. VIA [9]). Domains where highly optimized ORB messaging and transport protocols are particularly important include (1) multimedia applications running over high-speed networks, such as Gigabit Ethernet or ATM [21] and (2) real-time applications running over embedded system interconnects, such as VME or CompactPCI.

2.3 Solutions

To address the challenges outlined above, we have adopted a protocol-centric view to develop a high-performance and real-time ORB endsystem, which is shown in Figure 3. Our prior research on CORBA middleware has explored the efficiency, predictability, and scalability aspects of ORB endsystem design, including static [5] and dynamic [22] scheduling, event processing [23], synchronous [24] and asynchronous [11] ORB Core architectures, systematic benchmarking of multiple ORBs [25], and optimization principle patterns for ORB performance [26]. This paper extends our earlier work by focusing on the integration of the following topics: *(1) event-driven demultiplexing, (2) real-time I/O scheduling, (3) network protocol processing, and (4) inter-orb protocol processing.* Fig-

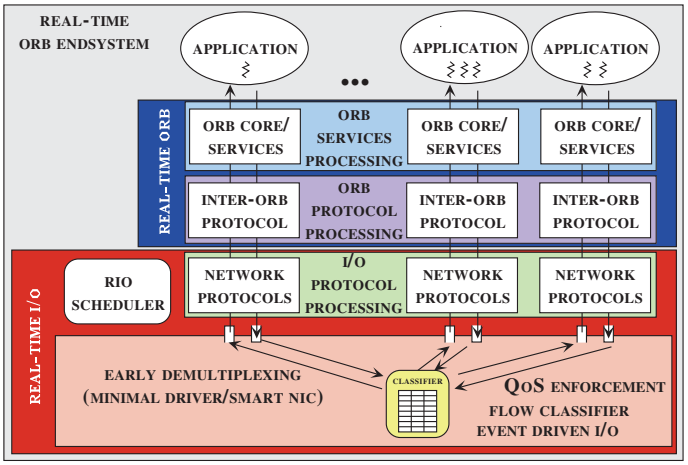

Figure 3: Architecture of a High-performance and Real-time ORB Endsystem

ure 3 deliberately does not delineate the protection boundary between the kernel and user domains. In principle, topics 2–4 can be performed in either domain and do not necessarily require privileged access to resources [27, 28].

In addition to identifying the horizontal layers of an ORB endsystem, Figure 3 also depicts its vertical partitions, where resources are dedicated to active connections. Each active connection is associated with a set of preallocated resources along a path [29] through the endsystem that ranges from network interface to application. We use this design strategy to ameliorate the effects of using shared queues and processing contexts [6].

Our solution addresses thread and packet-based priority inversions and inflexible inter-ORB protocols. These problems essentially are resource management issues on and between the endsystems. This can be addressed with a judicious use of preallocated resources, prioritizing I/O processing and providing the middleware with mechanisms to exploit optimized I/O features. Consequently, our solution must provide mechanisms for minimizing work performed with interrupt priorities, providing I/O processing threads in the kernel, preallocating memory and other I/O resources and facilities for inter-ORB protococols to exploit available optimizations in the underlying I/O subsystem.

Early demultiplexing: This feature is concerned with reducing unbounded priority inversion [30, 31, 15] by (1) minimizing the time spent processing packets with interrupt priorities and (2) associating all incoming and outgoing packets with preallocated resources. Incoming packets are demultiplexed and associated with the correct priorities and a specific Stream early in the packet processing sequence, *i.e.*, in the network interface driver [10]. RIO's design minimizes thread-based priority inversion by vertically integrating packets received at the network interface with the corresponding thread priorities in TAO's ORB Core. Section 3.1 describes how the TAO's RIO subsystem implements early demultiplexing.

Schedule-driven protocol processing: To minimize thread-based priority inversions, this feature performs all protocol processing with threads that are scheduled with the appropriate real-time priorities [32, 31, 14]. RIO's design schedules network interface bandwidth and CPU time to minimize priority inversion and decrease interrupt overhead during protocol processing. Section 3.2 describes how the TAO's RIO subsystem implements schedule-driven protocol processing.

Dedicated STREAMS: This feature addresses packet-based priority inversions by isolating request packets belonging to different priority groups to minimize FIFO queueing and shared resource locking overhead [33]. RIO's design alleviates resource conflicts that can otherwise cause thread-based and packet-based priority inversions. Section 3.3 describes how the TAO's RIO subsystem implements Dedicated STREAMS.

ORB Pluggable protocols framework: To address the limitations with inter-ORB protocol implementations outlined in Section 2.2.3, it must be possible for an ORB endsystem to add new protocol adaptors and exploit underlying hardware. Achieving this integration requires the ORB endsystem to satisfy the following seemingly contradictory goals:

- **Abstracting away from platform variation:** To maximize flexibility, application should be shielded from dependencies on specialized hardware or OS interfaces. In particular, applications should not require modifications when new platforms and communications links are configured. Thus, platform- and network-specific information should be encapsulated within the middleware framework

- **Leveraging custom platform features:** To maximize performance, applications that use middleware should benefit from specialized hardware, OS, and communication links available on a particular platform. For example, available network signaling and optimized network interface architectures may provide custom features, such as zero-copy I/O, bandwidth reservations, low latency connections, or optimized buffering strategies.

To achieve both these goals, we developed a highly extensible *pluggable protocols framework* [19] for TAO that presents a uniform, yet extensible, network programming interface. We use this framework to extend TAO's concurrency architecture and thread priority mechanisms into its RIO subsystem, thereby minimizing key sources of priority inversion that can cause non-determinism and jitter.

These features are augmented by RIO's zero-copy buffer management optimizations [7, 10]. These optimizations elim-

inate unnecessary data-copying overhead between application processes/threads, network protocol stacks, and high-performance network interfaces that support advanced I/O features, such as protected DMA, read/write directly to host memory, priority queues, programmable interrupts, and paced transmission. Section 4 presents TAO's pluggable protocols frameworks and Section 5 describes how it is used to integrate high-performance network interfaces with ORB endsystems.

3 Implementing TAO's Real-Time I/O (RIO) Subsystem for Solaris

This section describes the implementation of TAO's real-time I/O (RIO) subsystem [6] for Solaris over an ATM network. We selected Solaris to explore kernel space protocol implementations and architectural implementation, as well as to extend earlier work on NetBSD [27].

The RIO subsystem enhances the Solaris 2.5.1 OS kernel and a Fore ATM interface driver by providing QoS specification and enforcement features that complement TAO's highly predictable real-time concurrency and connection architecture [24]. Figure 4 presents the architectural components in the RIO subsystem and illustrates their relationship to other TAO ORB endsystem components.

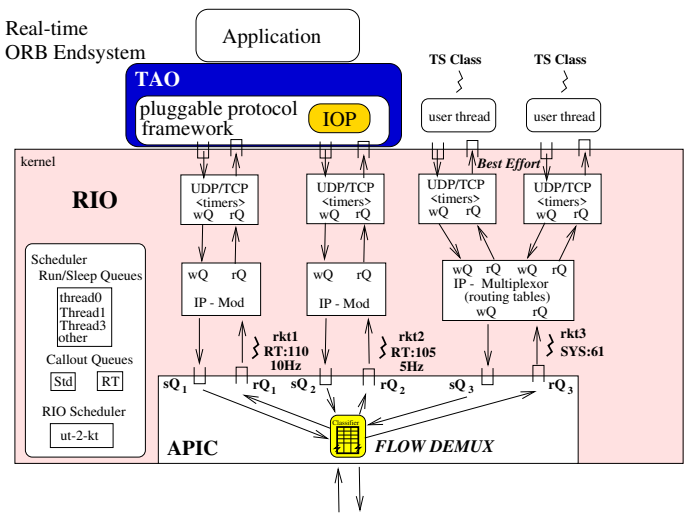

Figure 4: Architecture of the RIO Subsystem and Its Relationship to TAO

TAO's RIO subsystem is targeted currently for ATM/IP network interfaces, such as 155 Mbps FORE Systems SBA-200e ATM interfaces and 1.2 Gbps ATM Port Interconnect Controller (APIC) network interface. The APIC is particularly interesting because it supports optimized protocol development, zero-copy semantics, and real-time performance [7, 10]. However, RIO is designed to support other high-performance network interfaces, such as VIA [9], that provide similar QoS-enabled I/O features.

Below, we outline each of RIO's features, explain how they relate to features in the Solaris I/O subsystem, and justify our design and implementation decisions. Our discussion focuses on how we resolved the key design challenges faced when building the RIO subsystem.

3.1 Early Demultiplexing

Context: ATM is a connection-oriented network protocol that uses virtual circuits (VCs) to switch ATM cells at high speeds [34]. Each ATM connection is assigned a virtual circuit identifier (VCI)[3] that is included as part of the cell header.

Problem: In Solaris STREAMS, packets received by the ATM network interface driver are processed sequentially and passed up to the IP multiplexor in FIFO order. Therefore, any information containing the packets' priority or specific connection is lost.

Solution: The RIO subsystem uses a packet classifier [35] to exploit the early demultiplexing feature of ATM [10] by vertically integrating its ORB endsystem architecture, as shown in Figure 5. Early demultiplexing uses the VCI field in a request

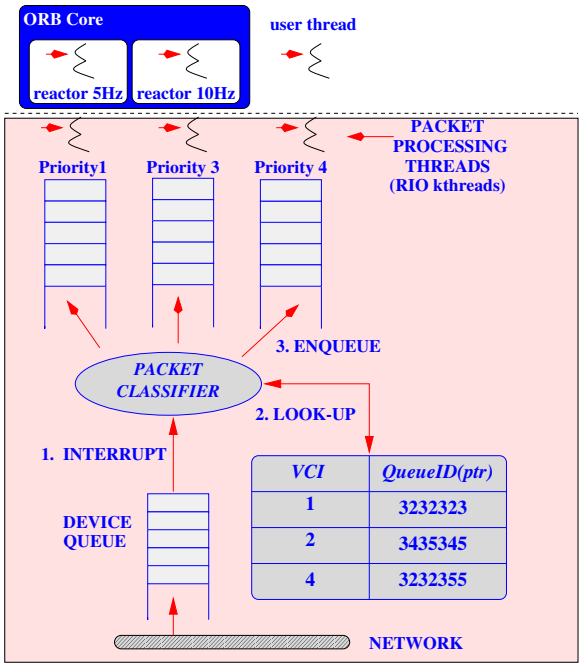

Figure 5: Early Demultiplexing in the RIO Subsystem

packet to determine its final destination thread efficiently.

[3] A virtual path identifier is also assigned, though we only consider the VCI in this paper.

Early demultiplexing helps alleviate packet-based priority inversion because packets need not be queued in FIFO order. Instead, RIO supports *priority-based queueing*, where packets destined for high-priority applications are delivered ahead of low-priority packets. In contrast, the Solaris default network I/O subsystem processes all packets at the same priority, regardless of the destination application thread.

Implementing early demultiplexing in RIO: The packet classifier in TAO's I/O subsystem can consult TAO's real-time scheduling service to determine where the packet should be placed. This is required when multiple applications use a single VC, as well as when the link layer is not ATM. In these cases, it is necessary to identify packets and associate them with rates/priorities on the basis of higher-level protocol addresses like TCP port numbers. Moreover, the APIC device driver can be modified to search the TAO's run-time scheduler [5] in the ORB's memory. TAO's run-time scheduler maps TCP port numbers to rate groups in constant $O(1)$ time.

At the lowest level of the RIO endsystem, the ATM driver distinguishes between packets based on their VCIs and stores them in the appropriate RIO queue (*rQ* for read queue and *wQ* for write queue). Each RIO queue pair is associated with exactly one Stream, but each Stream can be associated with zero or more RIO queues, *i.e.*, there is a many to one relationship for the RIO queues. The RIO protocol processing kernel thread (kthread) associated with the RIO queue then delivers the packets to TAO's ORB Core, as shown in Figure 4.

Figure 4 also illustrates how all periodic connections are assigned a dedicated Stream, RIO queue pair, and RIO kthread for input protocol processing. RIO kthreads typically service their associated RIO queues at the periodic rate specified by an application. In addition, RIO can allocate kthreads to process the output RIO queue.

3.2 Schedule-driven Protocol Processing

Context: Many real-time applications require periodic I/O processing [32]. For example, avionics mission computers must process sensor data periodically to maintain accurate situational awareness [23]. If the mission computing system fails unexpectedly, corrective action must occur immediately.

Problem: Protocol processing of input packets in Solaris STREAMS is *demand-driven*, *i.e.*, when a packet arrives the STREAMS I/O subsystem suspends all user-level processing and performs protocol processing on the incoming packet. Demand-driven I/O can incur priority inversion, such as when the incoming packet is destined for a thread with a priority lower than the currently executing thread. Thus, the ORB endsystem may fail to meet the QoS requirements of the higher priority thread.

When sending packets to another host, protocol processing is often performed within the context of the application thread that performed the `write` operation. The resulting packet is passed to the driver for immediate transmission on the network interface link. With ATM, a pacing value can be specified for each active VC, which allows simultaneous pacing of multiple packets out the network interface. However, pacing may not be adequate in overload conditions because output buffers can overflow, thereby losing or delaying high-priority packets.

Solution: RIO's solution is to perform *schedule-driven*, rather than demand-driven, protocol processing of network I/O requests. This solution *co-schedules* kernel-threads with real-time application threads to integrate a priority-based concurrency architecture vertically throughout the ORB endsystem. All protocol processing is performed in the context of kthreads that are scheduled with the appropriate real-time priorities.

Implementing Schedule-driven protocol processing in RIO: The RIO subsystem uses a *thread pool* [24] concurrency model to implement its schedule-driven kthreads. Thread pools are appropriate for real-time ORB endsystems because they (1) amortize thread creation run-time overhead and (2) place an upper limit on the total percentage of CPU time used by RIO kthread overhead.

Figure 6 illustrates the thread pool model used in RIO. This

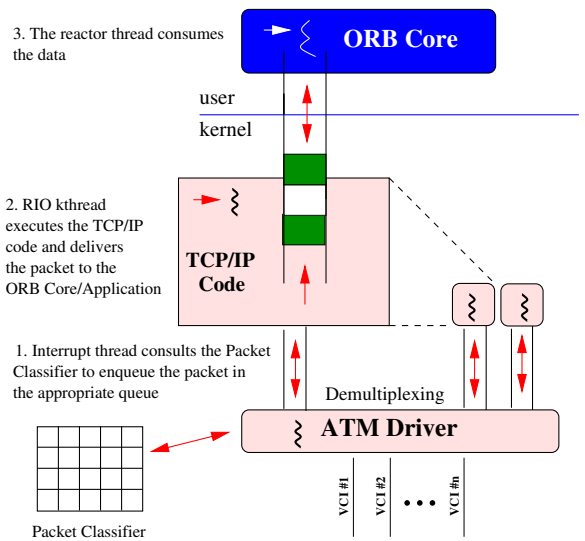

Figure 6: RIO Thread Pool Processing of TCP/IP with QoS Support

pool of protocol processing kthreads (RIO kthreads), is created at I/O subsystem initialization. Initially these threads are not bound to any connection and are inactive until needed.

Each kthread in RIO's pool is associated with a queue. The queue links the various protocol modules in a Stream. Each

thread is assigned a particular *rate*, based on computations from TAO's static scheduling service [5]. This rate corresponds to the frequency at which requests are specified to arrive from clients. Packets are placed in the queue by the application (for clients) or by the interrupt handler (for servers). Protocol code is then executed by the thread to shepherd the packet through the queue to the network interface card or up to the application.

Applications can use the standard real-time CORBA [36] middleware APIs provided by TAO to schedule network interface bandwidth and CPU time to minimize priority inversion and decrease interrupt overhead during protocol processing.

3.3 Dedicated Streams

Context: The RIO subsystem is responsible for enforcing QoS requirements for statically scheduled real-time applications with deterministic requirements.

Problem: Unbounded priority inversions can result when packets are processed in the I/O subsystem asynchronously, without respect for their priorities.

Solution: The effects of priority inversion in the I/O subsystem are minimized by isolating data paths through STREAMS to minimize resource contention. This is done in RIO by providing a *dedicated* STREAM connection path that (1) allocates separate buffers in the ATM driver and (2) associates kernel threads with the appropriate RIO scheduling priority for protocol processing. This design resolves resource conflicts that can otherwise cause thread-based and packet-based priority inversions.

Implementing Dedicated STREAMs in RIO: Figure 4 depicts our implementation of Dedicated STREAMS in RIO. Incoming packets are demultiplexed in the driver and passed to the appropriate Stream. A map in the driver's interrupt handler determines (1) the type of connection and (2) whether the packet should be placed on a queue or processed at interrupt context.

Typically, low-latency connections are processed in interrupt context. All other connections have their packets placed on the appropriate STREAM queue. Each queue has an associated protocol kthread that processes data through the Stream. These threads may have different priorities assigned by TAO's scheduling service.

A key feature of RIO's Dedicated STREAMS design is its use of multiple output queues in the client's ATM driver. With this implementation, each connection is assigned its own transmission queue in the driver. The driver services each transmission queue according to its associated priority. This design allows RIO to associate low-latency connections with high-priority threads to assure that its packets are processed before all other packets in the system.

4 Overview of TAO's Pluggable Protocols Framework

Simply providing enhancements to an I/O subsystem will not necessarily provide performance gains to applications built on top of middleware. Middleware provides transparency to many aspects of communication in order to isolate the application developers from inherent and accidental complexity associated with developing large, distributed applications [37]. However, in order to realize the full benefit of an optimized I/O subsystem and advanced network interfaces, the inter-ORB protocol processing components of the middleware must provide a facility for leveraging the I/O subsystem. We have implemented the ORB protocol processing components as a framework that allows for both novel protocol implementations and for application developers to specify protocol attributes.

4.1 The CORBA Protocol Interoperability Architecture

The CORBA specification [1] defines an architecture for ORB interoperability. Although a complete description of the model is beyond the scope of this paper, this section outlines the parts that are relevant to our present topic, *i.e.*, inter-ORB protocols for high-performance network interfaces and QoS-enabled I/O subsystems.

CORBA Inter-ORB Protocols (IOP)s define interoperability between ORB endsystems. IOPs provide data representation formats and ORB messaging protocol specifications that can be mapped onto standard and/or customized transport protocols. Regardless of the choice of ORB messaging or transport protocol, however, the standard CORBA programming model is exposed to the application developers. Figure 7 shows the relationships between these various components and layers.

In the CORBA protocol interoperability architecture, the standard General Inter-ORB Protocol (GIOP) is defined by the CORBA specification [1]. In addition, CORBA defines a transport-specific mapping of GIOP onto the TCP/IP protocol suite called the Internet Inter-ORB Protocol (IIOP). ORBs must support IIOP to be "interoperability compliant." Other mappings of GIOP onto different transport protocols are allowed by the specification, as are different inter-ORB protocols, known as Environment Specific Inter-ORB Protocols (ESIOP)s.

Regardless of whether GIOP or an ESIOP is used, a CORBA IOP must define a data representation, an ORB message format, an ORB transport protocol or transport protocol adapter, and an object addressing format.

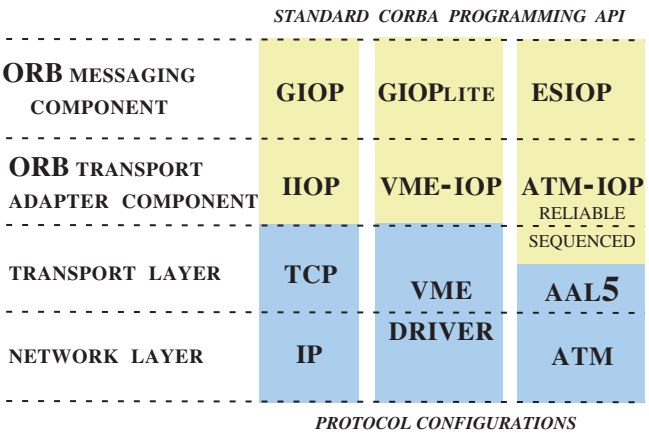

Figure 7: Relationship Between Inter-ORB Protocols and Transport-specific Mappings

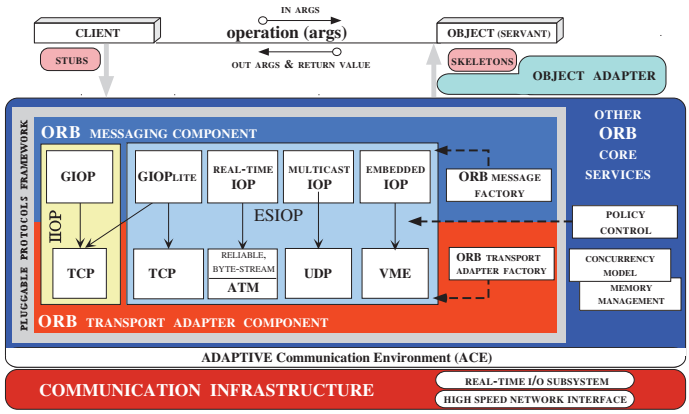

Figure 8: TAO's Pluggable Protocols Framework Architecture

4.2 TAO's Pluggable Protocols Framework Architecture

TAO's pluggable protocols framework allows custom ORB messaging and transport protocols to be configured flexibly and used transparently by CORBA applications. For example, if ORBs communicate over a high-performance networking protocol like ATM AAL5, then simpler, optimized ORB messaging and transport protocols can be configured to eliminate unnecessary features and overhead of the standard CORBA General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP). Likewise, TAO's pluggable protocols framework makes it straightforward to support customized embedded system interconnects, such as CompactPCI or VME, under standard CORBA inter-ORB protocols like GIOP.

To address the research challenges identified in Section 1.3, we identified logical communication component layers within TAO, factored out common features, defined general framework interfaces, and implemented components to support different concrete inter-ORB protocols. Higher-level components in the ORB, such as stubs, skeletons, and standard CORBA pseudo-objects, are decoupled from the implementation details of particular protocols, as shown in Figure 8. This decoupling is essential to resolve the various limitations with conventional ORBs outlined in Section 1.1 and discussed further in [19].

In general, the higher-level components in TAO use abstract interfaces to access the mechanisms provided by its pluggable protocols framework. Thus, applications can (re)configure custom protocols *without* requiring global changes to the ORB. Moreover, because applications typically access only the standard CORBA APIs, TAO's pluggable protocols framework is transparent to CORBA application developers.

Figure 8 also illustrates the key components in TAO's pluggable protocols framework: (1) the ORB messaging component, (2) the ORB transport adapter component, and (3) the ORB policy control component, which are outlined below.

4.2.1 ORB Messaging Component

This component is responsible for implementing ORB messaging protocols, such as the standard CORBA GIOP ORB messaging protocol, as well as custom ESIOPs. As described in [1], ORB messaging protocols should define a data representation, an ORB message format, an ORB transport protocol or transport adapter, and an object addressing format. Within this framework, ORB protocol developers are free to implement optimized Inter-ORB protocols and enhanced transport adaptors as long as the ORB interfaces are respected.

Each ORB messaging protocol implementation inherits from a common base class that defines a uniform interface. This interface can be extended to include new capabilities needed by special protocol-aware policies. For example, ORB end-to-end resource reservation or priority negotiation can be implemented in an ORB messaging component. TAO's pluggable protocols framework ensures consistent operational characteristics and enforces general IOP syntax and semantic constraints, such as error handling.

In general, it is not necessary to re-implement all aspects of an ORB messaging protocol. For example, TAO has a highly optimized presentation layer implementation that can be used by new protocols [26]. This presentation layer data representation contains well-tested and highly-optimized memory allocation strategies and data type translations. Thus, protocol developers can simply identify new memory or connection management strategies that can be used within the existing pluggable protocols framework.

Other key parts of TAO's ORB messaging component are its message factories. During connection establishment, these

factories instantiate objects that implement various ORB messaging protocols. These objects are associated with a specific connection and ORB transport adapter component, *i.e.*, the object that implements the component, for the duration of the connection.

4.2.2 ORB Transport Adapter Component

This component maps a specific ORB messaging protocol, such as GIOP or DCE-CIOP, onto a specific instance of an underlying transport protocol, such as TCP or ATM. Figure 8 shows an example in which TAO's transport adapter maps the GIOP messaging protocol onto TCP (this standard mapping is called IIOP). In this case, the ORB transport adapter combined with TCP corresponds to the transport layer in the Internet reference model. However, if ORBs are communicating over an embedded interconnect, such as a VME bus, the bus driver and DMA controller provide the "transport layer" in the communication infrastructure.

TAO's ORB transport component accepts a byte-stream from the ORB messaging component, provides any additional processing required, and passes the resulting data unit to the underlying communication infrastructure. Additional processing that can be implemented by protocol developers includes (1) concurrency strategies, (2) endsystem/network resource reservation protocols, (3) high-performance techniques, such as zero-copy I/O, shared memory pools, periodic I/O, and interface pooling, (4) enhancement of underlying communications protocols, *e.g.*, provision of a reliable byte-stream protocol over ATM, and (5) tight coupling between the ORB and efficient user-space protocol implementations, such as Fast Messages [38].

4.2.3 ORB Policy Control Component

This component allows applications to explicitly control the QoS attributes of configured ORB transport protocols. Since it is not possible to determine *a priori* all attributes defined by all protocols, an extensible *policy control* component is provided by TAO's pluggable protocols framework. TAO's policy control component implements the QoS framework defined in the CORBA Messaging [39] and Real-time CORBA [36] specifications.

To control the QoS attributes in the ORB, the CORBA QoS framework allows applications to specify various *policies*, such as buffer pre-allocations, fragmentation, bandwidth reservation, and maximum transport queue sizes. These policies can be set at the ORB-, thread-, or object-level, *i.e.*, application developers can set global policies that take effect for any request issued in a particular ORB. These global settings can be overridden on a per-thread, per-object, or even per-request basis. In general, the use of policies enables the CORBA specification to define semantic properties of ORB features precisely without (1) over-constraining ORB implementations or (2) increasing interface complexity for common use cases.

Certain policies, such as timeouts, can be shared between multiple protocols. Other policies, such as ATM virtual circuit bandwidth allocation, may apply to a single protocol. Each configured protocol can query TAO's policy control component to determine its policies and use them to configure itself for user needs. Moreover, a protocol implementation can simply ignore policies that do not apply to it.

TAO's policy control component also allows applications to select their protocol(s). This choice can be controlled by the `ClientProtocolPolicy` defined in the Real-time CORBA specification [36]. Using this policy, the application indicates its preferred protocol(s) and TAO's policy control component attempts to match that preference with the set of available protocols. Yet another policy controls the behavior of the ORB if an application's preferences cannot be satisfied, *e.g.*, either an exception is raised or another available protocol is selected transparently.

5 Integrating High-Performance Network Interfaces with ORB Endsystems

This section complements Section 3 and Section 4 by illustrating how TAO's RIO subsystem and pluggable protocols framework can be integrated with high-performance network interfaces. To focus the discussion, we present a use-case where ORB endsystems must support a high-performance, real-time CORBA application using the ATM Port Interconnect Controller (APIC) [7, 10] developed at Washington University. This scenario is based on our experience developing high-bandwidth, low-latency audio/video streaming applications [17] and avionics mission computing [23, 19].

5.1 High-performance Network Interface Features

As shown in Figure 9, the TAO ORB endsystem can be configured with a high-performance network interface and a real-time I/O (RIO) subsystem [6] designed to maximize available bandwidth to a mix of demanding applications. In this use-case, RIO is configured to support the 1.2 Gbps ATM Port Interconnect Controller (APIC) network interface.

The APIC is custom I/O chipset that incorporates several mechanisms designed to improve throughput and reduce latency. These mechanisms include (1) *zero-copy* shared memory pools between user- and kernel-space, (2) per-VC pacing, (3) two levels of priority queues, (4) interrupt disabling on a

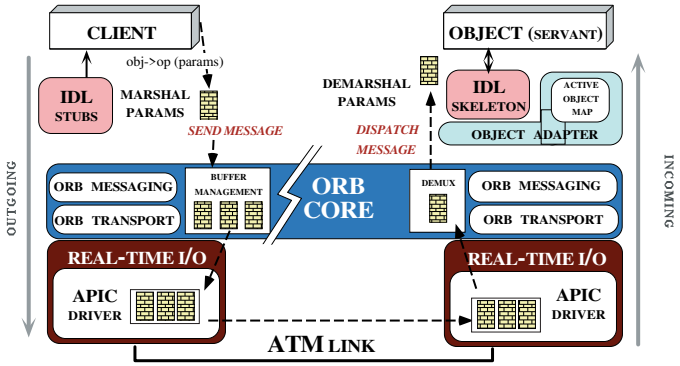

Figure 9: Real-Time ORB Endsystem Example

per-VC basis, and (5) *protected DMA*. The APIC's zero-copy mechanism [7] uses system memory to buffer cells, thereby minimizing on-board memory, which reduces its cost. The APIC's protected DMA [10] mechanism allows user-space protocols to queue buffers for transmission or reception to the network interface directly, thereby providing separate protected data channels to each active connection. To improve end-to-end throughput and latency, protected DMA bypasses intermediate kernel-level processing.

5.2 Multimedia Streaming Application Features

Multimedia applications running over high-performance networks require special optimizations to utilize available link bandwidth while reducing overall load on system resources such as memory and bus bandwidth. For example, consider Figure 10, where network interfaces supporting 1.2 Mbps

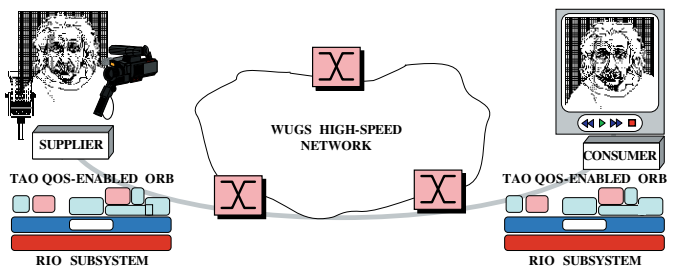

Figure 10: Example CORBA-based Multimedia Application

or 2.4 Mbps link speeds are used for a multimedia application based on the CORBA Audio/Video (A/V) Streaming Service [17].

In this scenario, we replaced GIOP/IIOP with a custom ORB messaging and transport protocol that transmits A/V frames using AAL5 over ATM to take full advantage of a high-performance ATM port interconnect controller (APIC) [10]. For example, applications can use the APICs features to establish network reservations that enforce their desired bandwidth and delay. Although the connection establishment and QoS negotiations are part of the underlying network protocol and the ORB's IOP, they will be transparent to the application.

5.3 Meeting ORB Endsystem Integration Design Challenges

Leveraging the underlying APIC network interface hardware to meet the end-to-end QoS requirements of the multimedia application described above necessitates the resolution of the following design challenges:

1. Custom protocols: This challenge centers on creating custom ORB messaging and transport protocols that can exploit the high-performance APIC network interface hardware. For the multimedia streaming application, a simple frame sequencing protocol can be used as an ESIOP. The goal is to simplify the messaging protocol, while adding any QoS related information to support the timely delivery of the video frames and audio. For example, an ORB message would correspond to one video frame or audio packet. A timestamp and sequence number can be sent along with each ORB message to facilitate synchronization between endpoints. The ORB messaging protocol can perform a similar function as the real-time protocol (RTP) and real-time control protocol (RTCP) [40].

This ORB messaging protocol can be mapped onto an ORB transport protocol using ATM AAL5. The ORB's transport adapter is then responsible for exploiting any local optimizations to hardware or OS I/O subsystem. For example, traditional ORB implementations will copy user parameters into ORB buffers used for marshaling. These may be allocated from global memory or possibly from a memory pool maintained by the ORB. In either case, at least one system call is required to obtain mutexes, allocate buffers and finally copy data. Thus, not only is an additional data copy incurred, but this scenario is rife with opportunities for priority inversion and indefinite application blocking.

2. Optimized protocol implementations: This challenge centers on optimizing communication protocol implementations, *e.g.*, by sharing memory between the application, TAO ORB middleware, RIO's I/O subsystem in the OS kernel, and the APIC network interface. This sharing can be achieved by requiring the message encapsulation process to use memory allocated from a common buffer pool [10, 26], which eliminates memory copies between user- and kernel-space when data is sent and received. The ORB endsystem manages this memory, thereby relieving application developers from this responsibility. In addition, the ORB endsystem can transparently manage the APIC interface driver, interrupt rates, and pacing parameters, as outlined in [6].

5.4 Bringing the Components Together

Figure 11 shows how the various ORB endsystem components described above can be configured together to support our example multimedia streaming application. In this configuration,

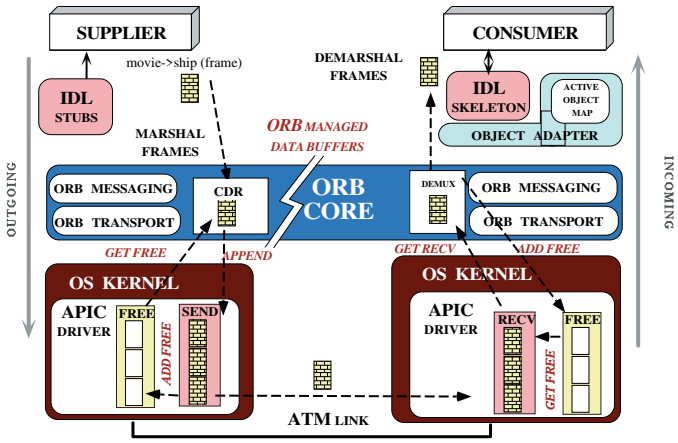

Figure 11: Shared Buffer Strategy

the ORB manages all the memory on behalf of the application. For instance, the application can request a set of buffers from the ORB, which it uses to send and receive video and audio data. TAO can be configured to allocate these buffers within memory shared between the application, ORB middleware, RIO subsystem, and APIC driver in the kernel. The ORB's transport adapter manages this shared buffer pool on a per-connection basis to minimize lock contention, *e.g.*, each active connection is assigned its own send and receive queues in Figure 11. Likewise, there are two free buffer pools per-connection, one for incoming packets and one for outgoing packets.

The ORB's pluggable protocols framework can ensure that only one application thread will be active within the send or receive operation of the transport adapter. Therefore, buffer allocation and de-allocation can be performed without extraneous locking [10]. Moreover, TAO's ORB endsystem configuration can be strategized so that application video and audio data can be copied conditionally into ORB buffers. For instance, it may be more efficient to copy relatively small data into ORB buffers, rather than use shared buffers between the ORB and the network interface. By using TAO's policy control component described in Section 4.2, applications can select their degree of sharing on a per-connection, per-thread, per-object, or per-operation basis.

6 ORB Endsystem Benchmarking Results

This section presents empirical results that show how the RIO subsystem decreases the upper bound on round-trip delay for low-latency applications and provides periodic processing guarantees for bandwidth-sensitive applications. The test systems are relatively slow compared to systems currently available however the relative speeds of the network interface, I/O bus and CPU remains valid. These empirical benchmarks that show how TAO's vertically integrated ORB endsystem, *i.e.*, its Object Adapter, ORB Core, pluggable protocols framework, RIO subsystem, and network interface, can (1) decrease the upper bound on round-trip delay for latency-sensitive applications and (2) provide periodic processing guarantees for bandwidth-sensitive applications. Earlier work has reported empirical results for separate benchmarks of each component in TAO's ORB endsystem, including the Object Adapter [6], ORB Core [24], pluggable protocols framework [19], and RIO subsystem [6, 15]. This section extends these results by highlighting the major features of TAO's ORB endsystem architecture that have been *integrated* with an optimized network interface and driver.

6.1 Hardware Configuration

Our experiments were conducted using a FORE Systems ASX-1000 ATM switch connected to two SPARCs: a uniprocessor 300 MHz UltraSPARC2 with 256 MB RAM and a 170 MHz SPARC5 with 64 MB RAM. Both SPARCs ran Solaris 2.5.1 and were connected via a FORE Systems SBA-200e ATM network interface to an OC3 155 Mbps port on the ASX-1000. The testbed configuration is shown in Figure 12.

6.2 Measuring the End-to-end Real-time Performance of the RIO Subsystem

Below, we present results that quantify the benefits gained in terms of bounded latency response times and periodic processing guarantees. RIO uses a periodic processing model to provide bandwidth guarantees and to bound maximum throughput on each connection.

6.2.1 Benchmarking Configuration

Our experiments were performed using the testbed configuration shown in Figure 12. To measure round-trip latency we use a client application that opens a TCP connection to an "echo server" located on the SPARC5. The client sends a 64 byte data block to the echo server, waits on the socket for data to return from the echo server, and records the round-trip latency.

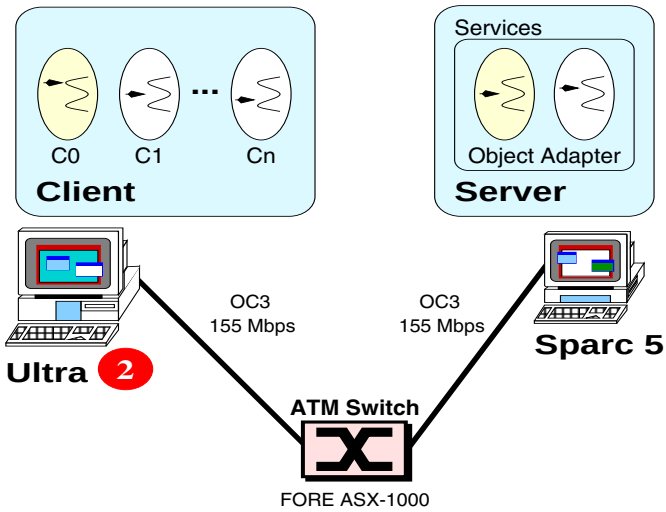

Figure 12: ORB Endsystem Benchmarking Testbed

The client application performs 10,000 latency measurements, then calculates the mean latency, standard deviation, and standard error. Both the client and server run at the same thread priority in the Solaris real-time (RT) scheduling class.

Bandwidth tests were conducted using a modified version of `ttcp` [41] that sent 8 KB data blocks over a TCP connection from the UltraSPARC2 to the SPARC5. Threads that receive bandwidth reservations are run in the RT scheduling class, whereas best-effort threads run in the TS scheduling class.

The default behavior of the Solaris I/O subsystem is to perform network protocol processing at interrupt context [15]. Our measurements reveal the effect of performing network protocol processing at interrupt context versus performing it in a RIO kthread. With the interrupt processing model, the input packet is processed immediately up through the network protocol stack. Conversely, with the RIO kthreads model, the packet is placed in a RIO queue and the interrupt thread exits. This causes a RIO kthread to wake up, dequeue the packet, and perform protocol processing within its thread context.

A key feature of using RIO kthreads for protocol processing is their ability to assign appropriate kthread priorities and to defer protocol processing for lower priority connections. Thus, if a packet is received on a high-priority connection, the associated kthread will preempt lower priority kthreads to process the newly received data.

Our previous results [6] revealed that using RIO kthreads in the RT scheduling class results in a slight increase of 13-15 μs in the round-trip processing times in our testing environment. This latency increase stems from RIO kthread dispatch latencies and queuing delays. However, the significant result was the overall reduction in latency jitter for real-time RIO kthreads.

6.2.2 Measuring Low-latency Connections with Competing Traffic

Benchmark design: This experiment measures the determinism of the RIO subsystem while performing prioritized protocol processing on a heavily loaded server. The results illustrate how RIO behaves when network I/O demands exceed the ability of the ORB endsystem to process all requests. The SPARC5 is used as the server in this test because it can process only ~75% of the full link speed on an OC3 ATM interface using `ttcp` with 8 KB packets.

Two different classes of data traffic are created for this test: (1) a low-delay, high-priority message stream and (2) a best-effort (low-priority) bulk data transfer stream. The message stream is simulated using the latency application described in Section 6.2.1. The best-effort, bandwidth intensive traffic is simulated using a modified version of the `ttcp` program, which sends 8 KB packets from the client to the server.

The latency experiment was first run with competing traffic using the default Solaris I/O subsystem. Next, the RIO subsystem was enabled, RIO kthreads and priorities were assigned to each connection, and the experiment was repeated. The RIO kthreads used for processing the low-delay, high-priority messages were assigned a real-time global priority of 100. The latency client and echo server were also assigned a real-time global priority of 100.

The best-effort bulk data transfer application was run in the time-sharing class. The corresponding RIO kthreads ran in the system scheduling class with a global priority of 60. In general, all best effort connections use a RIO kthread in the SYS scheduling class with a global priority of 60. Figure 13 shows the configuration for the RIO latency benchmark.

Benchmark results and analysis: The results from collecting 1,000 samples in each configuration are summarized in the table below:

	Mean	**Max**	**Min**	**Jitter**
Default	1072 μs	3158 μs	594 μs	497 μs
RIO	946 μs	2038 μs	616 μs	282 μs

This table compares the behavior of the default Solaris I/O subsystem with RIO. It illustrates how RIO lowers the upper bound on latency for low-delay, high-priority messages in the presence of competing network traffic. In particular, RIO lowered the maximum round-trip latency by 35% (1,120 μs), the average latency by 12% (126 μs), and jitter by 43% (215 μs). The distribution of samples are shown in Figure 14. This figure highlights how RIO lowers the upper bound of the round-trip latency values.

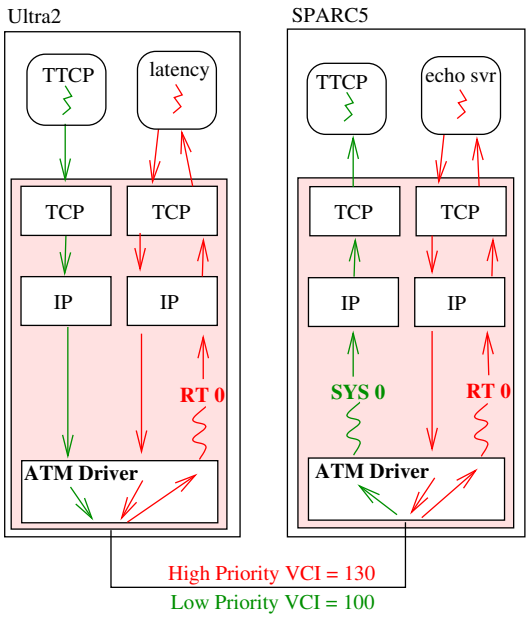

Figure 13: RIO Low-latency Benchmark Configuration

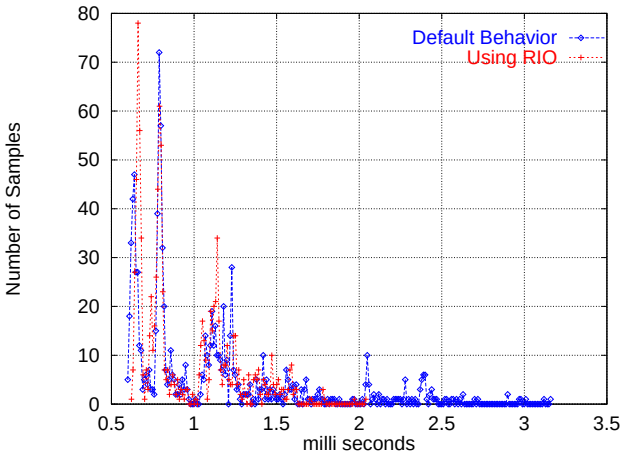

Figure 14: Latency with Competing Traffic

These performance results are particularly relevant for real-time systems where ORB endsystem predictability is crucial. The ability to specify and enforce end-to-end priorities over transport connections helps ensure that ORB endsystems achieve end-to-end determinism.

Another advantage of RIO's ability to preserve end-to-end priorities is that the overall system utilization can be increased. For instance, the experiment above illustrates how the upper bound on latency was reduced by using RIO to preserve end-to-end priorities. For example, system utilization may be unable to exceed 50% while still achieving a 2 ms upper bound for high-priority message traffic. However, higher system utilization can be achieved when an ORB endsystem supports real-time I/O. The results in this section demonstrate this: RIO achieved latencies no greater than 2.038 ms, even when the ORB endsystem was heavily loaded with best-effort data transfers.

Figure 15 shows the average bandwidth used by the modified `ttcp` applications during the experiment. The dip in

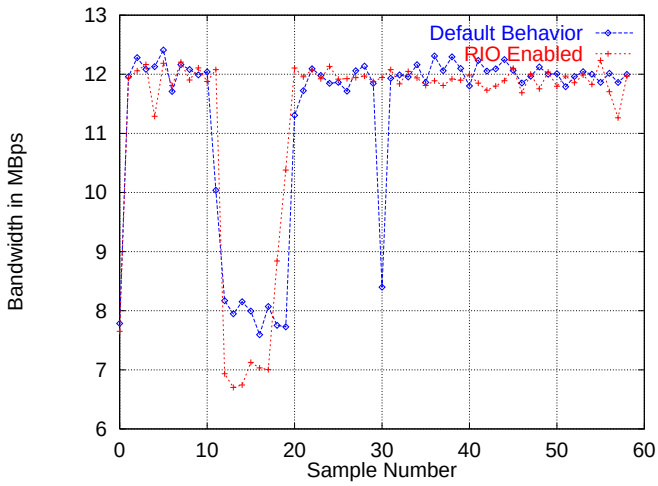

Figure 15: Bandwidth of Competing Traffic

throughput between sample numbers 10 and 20 occurred when the high-priority latency test was run, which illustrates how RIO effectively reallocates resources when high-priority message traffic is present. Thus, the best-effort traffic obtains slightly lower bandwidth when RIO is used.

6.2.3 Measuring Bandwidth Guarantees for Periodic Processing

Benchmark design: RIO can enforce bandwidth guarantees because it implements a schedule-driven protocol processing model [6], which *co-schedules* kernel threads with real-time application threads in the TAO's ORB Core. In contrast, the default Solaris I/O subsystem processes all input packets on-demand at interrupt context, *i.e.*, with a priority higher than all other application threads and non-interrupt kernel threads.

The following experiment demonstrates the advantages and accuracy of RIO's periodic protocol processing model. The experiment was conducted using three threads that receive specific periodic protocol processing, *i.e.*, bandwidth, guarantees from RIO. A fourth thread sends data using only best-effort guarantees.

All four threads run the `ttcp` program, which sends 8 KB data blocks from the UltraSPARC2 to the SPARC5. For each bandwidth-guaranteed connection, a RIO kthread was allocated in the real-time scheduling class and assigned appropriate periods and packet counts, *i.e.*, computation time. The best-effort connection was assigned the default RIO kthread,

which runs with a global priority of 60 in the system scheduling class. Thus, there were four RIO kthreads, three in the real-time scheduling class and one in the system class. The following table summarizes the RIO kthread parameters for the bandwidth experiment.

RIO Config	**Period**	**Priority**	**Packets**	**Bandwidth**
kthread 1	10 ms	110	8	6.4 MBps
kthread 2	10 ms	105	4	3.2 MBps
kthread 3	10 ms	101	2	1.6 MBps
kthread 4 (best-effort)	Async	60	Available	Available

The three application threads that received specific bandwidth guarantees were run with the same real-time global priorities as their associated RIO kthreads. These threads were assigned priorities related to their guaranteed bandwidth requirements – the higher the bandwidth the higher the priority. The `ttcp` application thread and associated RIO kthread with a guaranteed 6.4 MBps were assigned a real-time priority of 110. The application and RIO kernel threads with a bandwidth of 3.2 MBps and 1.6 MBps were assigned real-time priorities of 105 and 101, respectively.

RIO kthreads are awakened at the beginning of each period. They first check their assigned RIO queue for packets. After processing their assigned number of packets they sleep waiting for the start of the next period.

The best-effort application thread runs in the time sharing class. Its associated RIO kthread, called the "best-effort" RIO kthread, is run in the system scheduling class with a global priority of 60. The best-effort RIO kthread is not scheduled periodically. Instead, it waits for the arrival of an eligible network I/O packet and processes it "on-demand." End-to-end priority is maintained, however, because the best-effort RIO kthread has a global priority lower than either the application threads or RIO kthreads that handle connections with bandwidth guarantees.

Benchmark results and analysis: In the experiment, the best-effort connection starts first, followed by the 6.4 MBps, 3.2 MBps, and 1.6 MBps guaranteed connections, respectively. Figure 16 presents the results, showing the effect of the guaranteed connection on the best-effort connection.

This figure clearly shows that the guaranteed connections received their requested bandwidths. In contrast, the best-effort connection loses bandwidth proportional to the bandwidth granted to guaranteed connections. The measuring interval was small enough for TCPs "slow start" algorithm [42] to be observed.

Periodic protocol processing is useful to guarantee bandwidth and bound the work performed for any particular connection. For example, we can specify that the best-effort connection in the experiment above receive no more than 40% of the available bandwidth on a given network interface.

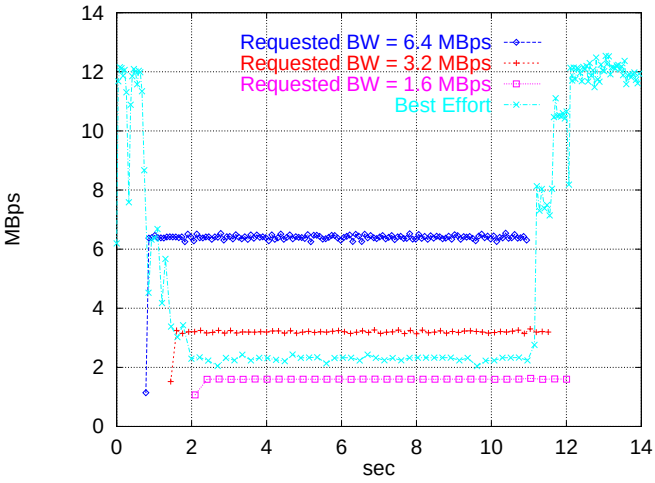

Figure 16: Bandwidth Guarantees in RIO

6.3 Measuring the End-to-end Real-time Performance of the TAO/RIO ORB Endsystem

Section 6.2 measured the performance of the RIO subsystem in isolation. This section combines RIO and TAO to create a vertically integrated real-time ORB endsystem and then measures the impact on end-to-end performance when run with prototypical real-time ORB application workloads [24].

6.3.1 Benchmark Design

The benchmark outlined below was performed twice: (1) without RIO, *i.e.*, using the unmodified default Solaris I/O subsystem and (2) using our RIO subsystem enhancements. Both benchmarks recorded average latency and the standard deviation of the latency values, *i.e.*, jitter. The server and client benchmarking configurations are described below.

Server benchmarking configuration: As shown in Figure 12, the server host is the 170 MHz SPARC5. This host runs the real-time ORB with two servants in the Object Adapter. The *high-priority* servant runs in a thread with an RT priority of 130. The *low-priority* servant runs in a lower priority thread with an RT thread priority of 100. Each thread processes requests sent to it by the appropriate client threads on the UltraSPARC2. The SPARC5 is connected to a 155 Mpbs OC3 ATM interface so the UltraSPARC2 can saturate it with network traffic.

Client benchmarking configuration: As shown in Figure 12, the client is the 300 MHz, uni-processor UltraSPARC2, which runs the TAO real-time ORB with one high-priority

client C_0 and n low-priority clients, $C_1 \ldots C_n$. The high-priority client is assigned an RT priority of 130, which is the same as the high-priority servant. It invokes two-way CORBA operations at a rate of 20 Hz.

All low-priority clients have the same RT thread priority of 100, which is the same as the low-priority servant. They invoke two-way CORBA operations at 10 Hz. In each call the client thread sends a value of type `CORBA::Octet` to the servant. The servant cubes the number and returns the result.

The benchmark program creates all the client threads at startup time. The threads block on a barrier lock until all client threads complete their initialization. When all threads inform the main thread that they are ready, the main thread unblocks the clients. The client threads then invoke 4,000 CORBA two-way operations at the prescribed rates.

RIO subsystem configuration: When the RIO subsystem is used, the benchmark has the configuration shown in Figure 17. With the RIO subsystem, high- and low-priority requests are

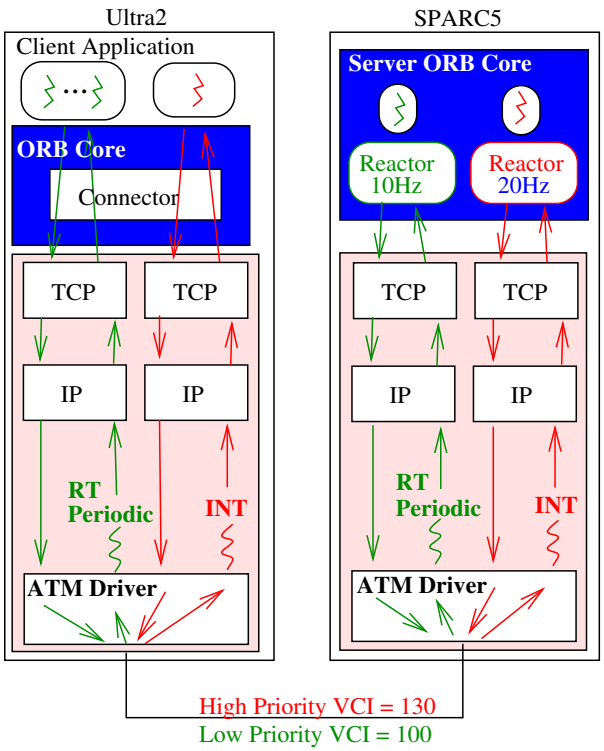

Figure 17: ORB Endsystem Benchmarking Configuration

treated separately throughout the ORB and I/O subsystem.

Low-priority client threads transmit requests at 10 Hz. There are several ways to configure the RIO kthreads. For instance, we could assign one RIO kthread to each low-priority client. However, the number of low-priority clients varies from 0 to 50. Plus all clients have the same period and send the same number of requests per period, so they have the same priorities. Thus, only one RIO kthread is used. Moreover, because it is desirable to treat low-priority messages as best-effort traffic, the RIO kthread is placed in the system scheduling class and assigned a global priority of 60.

To minimize latency, high-priority requests are processed by threads in the Interrupt (INTR) scheduling class. Therefore, we create two classes of packet traffic: (1) low-latency, high priority and (2) best-effort latency, low-priority. The high-priority packet traffic preempts the processing of any low-priority messages in the I/O subsystem, ORB Core, Object Adapter, and/or servants.

6.3.2 Benchmark Results and Analysis

This experiment shows how RIO increases overall determinism for high-priority, real-time applications without sacrificing the performance of best-effort, low-priority, and latency-sensitive applications. RIO's impact on overall determinism of the TAO ORB endsystem is shown by the latency and jitter results for the high-priority client C_0 and the average latency and jitter for 0 to 49 low-priority clients, $C_1 \ldots C_n$.

Figure 18 illustrates the average latency results for the high- and low-priority clients both with and without RIO. This figure

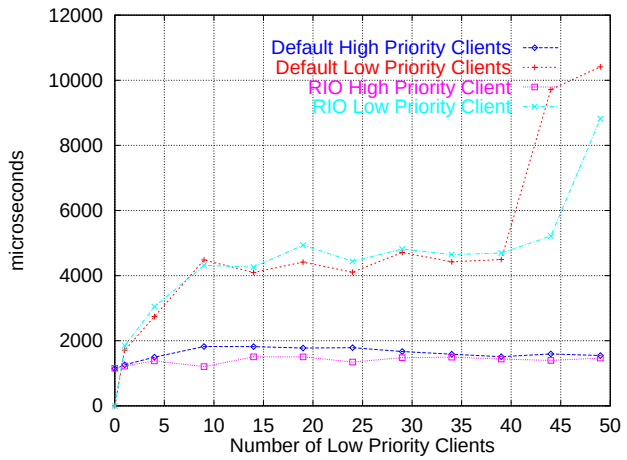

Figure 18: Measured Latency for All Clients with and without RIO

shows how TAO eliminates many sources of priority inversion within the ORB. Thus, high-priority client latency values are relatively constant, compared with low-priority latency values. Moreover, the high-priority latency values decrease when the the RIO subsystem is enabled. In addition, the low-priority clients' average latency values track the default I/O subsystems behavior, illustrating that RIO does not unduly penalize best-effort traffic. At 44 and 49 low-priority clients the RIO-enabled endsystem outperforms the default Solaris I/O subsystem.

Figure 19 presents a finer-grained illustration of the round-trip latency and jitter values for high-priority client vs. the number of competing low-priority clients. This figure illus-

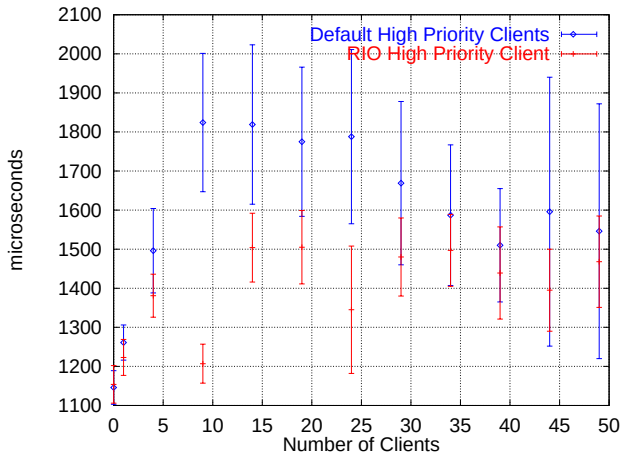

Figure 19: High-priority Client Latency and Jitter

trates how not only did RIO decrease average latency, but its jitter results were substantially better, as shown by the error bars in the figure. The high-priority clients averaged a 13% reduction in latency with RIO. Likewise, jitter was reduced by an average of 51%, ranging from a 12% increase with no competing low-priority clients to a 69% reduction with 44 competing low-priority clients.

In general, RIO reduced average latency and jitter because it used RIO kthreads to process low-priority packets. Conversely, in the default Solaris STREAMS I/O subsystem, servant threads are more likely to be preempted because threads from the INTR scheduling class are used for all protocol processing. Our results illustrate how this preemption can significantly increase latency and jitter values.

Figure 20 shows the average latency of low-priority client threads. This figure illustrates that the low-priority clients incurred no appreciable change in average latency. There was a slight increase in jitter for some combinations of clients due to the RIO kthreads dispatch delays and preemption by the higher priority message traffic. This result demonstrates how the RIO design can enhance overall end-to-end predictability for real-time applications while maintaining acceptable performance for traditional, best-effort applications.

7 Related Work

High-performance and real-time ORB endsystems are an emerging field of study. We have used TAO to research key dimensions of ORB endsystem design including static [5] and dynamic [22] scheduling, request demultiplexing [26], event processing [23], ORB Core connection and concurrency ar-

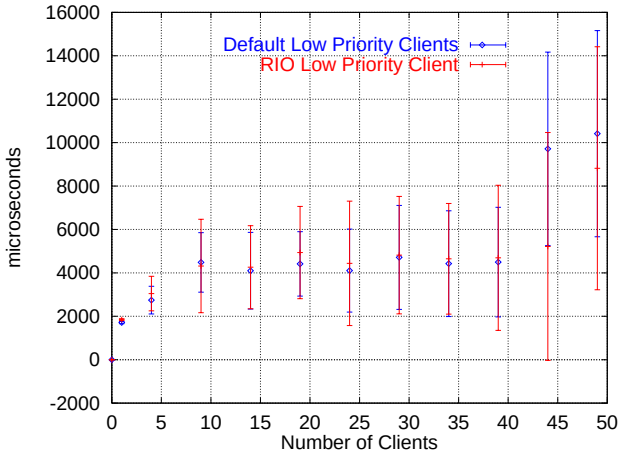

Figure 20: Low-priority Client Latency

chitectures [24], and IDL compiler stub/skeleton optimizations [26]. This paper extends results in [6, 19] to illustrate how the TAO's real-time I/O subsystem and pluggable protocols framework can exploit underlying hardware and software optimizations for high-performance network interfaces. The remainder of this section compares our work on TAO with related ORB endsystem research.

7.1 Related Work on QoS-enabled I/O Subsystems

Our real-time I/O (RIO) subsystem incorporates advanced techniques [8, 10, 31, 33, 14] for high-performance and real-time protocol implementations. Below, we compare RIO with related work on I/O subsystems.

I/O subsystem support for QoS: The Scout OS [29] employs the notion of a *path* to expose the state and resource requirements of all processing components in a *flow*. Similarly, our RIO subsystem reflects the path principle and incorporates it with TAO and Solaris to create a vertically integrated real-time ORB endsystem. For instance, RIO subsystem resources like CPU, memory, and network interface and network bandwidth are allocated to an application-level connection/thread during connection establishment, which is similar to Scout's binding of resources to a path.

Scout represents a fruitful research direction, which is complementary with our emphasis on demonstrating similar capabilities in existing operating systems, such as Solaris and NetBSD. At present, paths have been used in Scout largely for MPEG video decoding and display and not for protocol processing or other I/O operations. In contrast, we have successfully used RIO for a number of real-time avionics applications [23] with deterministic QoS requirements.

SPIN [43, 44] provides an extensible infrastructure and a

core set of extensible services that allow applications to safely change the OS interface and implementation. Application-specific protocols are written in a type-safe language, *Plexus*, and configured dynamically into the SPIN OS kernel. Because these protocols execute within the kernel, they can access network interfaces and other OS system services efficiently. To the best of our knowledge, however, SPIN does not support end-to-end QoS guarantees.

Enhanced I/O subsystems: Other related research has focused on enhancing performance and fairness of I/O subsystems, though not specifically for the purpose of providing real-time QoS guarantees. These techniques are directly applicable to designing and implementing real-time I/O and providing QoS guarantees, however, so we compare them with our RIO subsystem below.

[33] applies several high-performance techniques to a STREAMS-based TCP/IP implementation and compares the results to a BSD-based TCP/IP implementation. This work is similar to RIO, because Roca and Diot parallelize their STREAMS implementation and use early demultiplexing and dedicated STREAMS, known as Communication Channels (CC). The use of CC exploits the built-in flow control mechanisms of STREAMS to control how applications access the I/O subsystem. This work differs from RIO in that it focuses entirely on performance issues and not sources of priority inversion. For example, minimizing protocol processing in an interrupt context is not addressed.

[14, 31] examines the effect of protocol processing with interrupt priorities and the resulting priority inversions and livelock [14]. Both approaches focus on providing fairness and scalability under network load. In [31], a network I/O subsystem architecture called *lazy receiver processing* (LRP) is used to provide stable overload behavior. LRP uses early demultiplexing to classify packets, which are then placed into per-connection queues or on network interface channels. These channels are shared between the network interface and OS. Application threads read/write from/to network interface channels so input and output protocol processing is performed in the context of application threads. In addition, a scheme is proposed to associate kernel threads with network interface channels and application threads in a manner similar to RIO. However, LRP does not provide QoS guarantees to applications.

[14] proposed a somewhat different architecture to minimize interrupt processing for network I/O. They propose a polling strategy to prevent interrupt processing from consuming excessive resources. This approach focuses on scalability under heavy load. It did not address QoS issues, however, such as providing per-connection guarantees for fairness or bandwidth, nor does it charge applications for the resources they use. It is similar to our approach, however, in that (1) interrupts are recognized as a key source of non-determinism and (2) schedule-driven protocol processing is proposed as a solution.

While RIO shares many elements of the approaches described above, we have combined these concepts to create the first vertically integrated real-time ORB endsystem. The resulting ORB endsystem provides scalable performance, periodic processing guarantees and bounded latency, as well as an end-to-end solution for real-time distributed object computing middleware and applications.

7.2 Related Work on Pluggable Protocol Frameworks

The design of TAO's pluggable protocols framework is influenced by prior research on the design and optimization of protocol frameworks for communication subsystems, as described below.

Configurable communication frameworks: The *x*-kernel [45], System V STREAMS [46], Conduit+ [47], ADAPTIVE [48], and F-CSS [49] are all configurable communication frameworks that provide a protocol backplane consisting of standard, reusable services that support network protocol development and experimentation. These frameworks support flexible composition of modular protocol processing components, such as connection-oriented and connectionless message delivery and routing, based on uniform interfaces.

The frameworks for communication subsystems listed above focus on implementing various protocol layers beneath relatively low-level programming APIs, such as sockets. In contrast, TAO's pluggable protocols framework focuses on implementing and/or adapting to transport protocols beneath a higher-level OO middleware API, *i.e.*, the standard CORBA programming API. Therefore, existing communication subsystem frameworks can provide building block protocol components for TAO's pluggable protocols framework.

CORBA pluggable protocol frameworks: The architecture of TAO's pluggable protocols framework is based on the ORBacus Open Communications Interface (OCI) [50]. The OCI framework provides a flexible, intuitive, and portable interface for pluggable protocols. The framework interfaces are defined in IDL, with a few special rules to map critical types, such as data buffers.

Defining pluggable protocol interfaces with IDL permits developers to familiarize themselves with a single programming model that can be used to implement protocols in different languages. In addition, the use of IDL makes possible to write pluggable protocols that are portable among different ORB implementations and platforms.

Though the OCI pluggable protocols frameworks is useful for many applications and ORBs, TAO implements a highly

optimized pluggable protocol framework that is tuned for high-performance and real-time application requirements. For example, TAO's pluggable protocols framework can be integrated with zero-copy high-speed network interfaces [10, 8, 6, 20], embedded systems [23], or high-performance communication infrastructures like Fast Messages [38].

8 Concluding Remarks

To be an effective platform for performance-sensitive applications, ORB endsystems must preserve communication layer QoS properties to applications end-to-end. It is essential, therefore, to define a *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, end-to-end) integrated high-performance ORB endsystem. This paper presents the design and performance of such an ORB endsystem, called TAO, which provides a pluggable protocols framework to leverage high-performance network interfaces and real-time I/O (RIO) subsystems.

TAO's pluggable protocols framework provides an integrated set of (1) connection concurrency strategies, (2) endsystem/network resource reservation protocols, (3) high-performance techniques, such as zero-copy I/O, shared memory pools, periodic I/O, and interface pooling, that can be used to integrate applications with high-performance I/O subsystem and protocol implementations. The RIO subsystem enhances the Solaris 2.5.1 kernel to enforce the QoS features of the TAO ORB endsystem. RIO supports a vertically integrated, high-performance ORB endsystem that provides three classes of I/O, best-effort, periodic and low latency, which can be used to (1) increase throughput, (2) decrease latency, and (3) improve end-to-end predictability. In addition, RIO supports periodic protocol processing, guarantees I/O resources to applications, and minimizes the effect of flow control within each Stream.

A novel feature of the RIO subsystem and TAO's pluggable protocols framework is the integration of real-time scheduling and protocol processing, which allows TAO to support guaranteed bandwidth and low-delay applications. To accomplish this, we extended TAO's real-time concurrency architecture and thread priority mechanisms into RIO. This design minimizes sources of priority inversion that can cause nondeterminism and jitter.

The following are the key lessons we learned from our integration of RIO with TAO and its pluggable protocol framework:

Vertical integration of ORB endsystems is essential for end-to-end priority preservation: Conventional operating systems and ORBs do not provide adequate support for the QoS requirements of distributed, real-time applications [6, 15]. Meeting these needs requires a vertically integrated ORB endsystem that can deliver end-to-end QoS guarantees at multiple levels. The ORB endsystem described in this paper addresses this need by combining a real-time I/O (RIO) subsystem with the TAO ORB Core [24] and Object Adapter [26], which are designed explicitly to preserve end-to-end QoS properties in distributed real-time systems. RIO is designed to operate with high-performance interfaces such as the 1.2 Gbps ATM port interconnect controller (APIC) [10].

Schedule-driven protocol processing reduces jitter significantly: After integrating RIO with TAO, we measured a significant reduction in average latency and jitter. Moreover, the latency and jitter of low-priority traffic were not affected adversely. Our results illustrate how configuring asynchronous protocol processing [32] strategies in the Solaris kernel can provide significant improvements in ORB endsystem behavior, compared with the conventional Solaris I/O subsystem. As a result of our RIO enhancements to Solaris, TAO is the first standards-based, ORB endsystem to support end-to-end QoS guarantees over ATM/IP networks [34].

Input livelock is a dominant source of ORB endsystem non-determinism: During the development and experimentation of RIO, we observed that the dominant source of nondeterminism was *input livelock* [14], which degrades overall endsystem performance by processing all incoming packets at interrupt context. In particular, priority inversion resulting from processing all input packets at interrupt context is unacceptable for many real-time applications. Using RIO kthreads for input packet processing yielded the largest gain in overall system predictability. The underscores the importance of integrating high-performance network interfaces with real-time middleware and I/O subsystems in order to minimize priority inversions.

Future RIO research is focusing on integrating other OS platforms and network interfaces, as well as exporting a standardized programming API to higher-level ORB middleware. We continue to enhance TAO's pluggable protocol framework [19] to support new ORB messaging, transport protocols, and platforms. The TAO research effort has influenced the OMG Real-time CORBA specification [36]. The C++ source code for TAO and the benchmarks presented in Section 6 is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. The RIO subsystem is available to Solaris source licensees.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[2] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[3] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

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

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

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

[7] 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.

[8] T. v. Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *15th ACM Symposium on Operating System Principles*, ACM, December 1995.

[9] Compaq, Intel, and Microsoft, "Virtual Interface Architecture, Version 1.0." http://www.viarch.org, 1997.

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

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

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

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

[14] J. C. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driver Kernel," in *Proceedings of the USENIX 1996 Annual Technical Conference*, (San Diego, CA), USENIX, Jan. 1996.

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

[16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[17] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

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

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

[20] R. S. Madukkarumukumana and H. V. Shah and C. Pu, "Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks," in *Proceedings of the 2nd Usenix Windows NT Symposium*, August 1998.

[21] Vishal Kachroo, Yamuna Krishnamurthy, Fred Kuhns, Ronald G. Akers, Pradeep Avasthi, Surender Kumar, and Vidya Narayanan, "Design and Implementation of QoS enabled OO Middleware," in *Internet2 QoS Workshop*, February 2000.

[22] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000.

[23] 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.

[24] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2000.

[25] 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.

[26] 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.

[27] R. Gopalakrishnan and G. M. Parulkar, "Efficient User Space Protocol Implementations with QoS Guarantees using Real-time Upcalls," Tech. Rep. 96-11, Washington University Department of Computer Science, March 1996.

[28] R. Gopalakrishnan and G. Parulkar, "A Real-time Upcall Facility for Protocol Processing with QoS Guarantees," in 15^{th} *Symposium on Operating System Principles (poster session)*, (Copper Mountain Resort, Boulder, CO), ACM, Dec. 1995.

[29] D. Mosberger and L. Peterson, "Making Paths Explicit in the Scout Operating System," in *Proceedings of OSDI '96*, Oct. 1996.

[30] R. Gopalakrishnan and G. Parulkar, "Quality of Service Support for Protocol Processing Within Endsystems," in *High-Speed Networking for Multimedia Applications* (W. Effelsberg, *et al.*, ed.), Kluwer Academic Publishers, 1995.

[31] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," in *Proceedings of the* 1^{st} *Symposium on Operating Systems Design and Implementation*, USENIX Association, October 1996.

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

[33] T. B. Vincent Roca and C. Diot, "Demultiplexed Architectures: A Solution for Efficient STREAMS-Based Communication Stacks," *IEEE Network Magazine*, vol. 7, July 1997.

[34] G. Parulkar, D. C. Schmidt, and J. S. Turner, "$\mathrm{a^{I}t^{P}m}$: a Strategy for Integrating IP with ATM," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.

[35] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. L. Peterson, "Pathfinder: A pattern-based packet classifier," in *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*, USENIX Association, November 1994.

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

[37] D. C. Schmidt, "Using Design Patterns to Develop High-Performance Object-Oriented Communication Software Frameworks," in *Proceedings of the 8th Annual Software Technology Conference*, Apr. 1996.

[38] M. Lauria, S. Pakin, and A. Chien, "Efficient Layering for High Speed Communication: Fast Messages 2.x.," in *Proceedings of the 7th High Performance Distributed Computing (HPDC7) conference*, (Chicago, Illinois), July 1998.

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

[40] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications," *Network Information Center RFC 1889*, January 1996.

[41] USNA, *TTCP: a test of TCP and UDP Performance*, Dec 1984.

[42] W. R. Stevens, *TCP/IP Illustrated, Volume 1*. Reading, Massachusetts: Addison Wesley, 1993.

[43] B. Bershad, "Extensibility, Safety, and Performance in the Spin Operating System," in *Proceedings of the* 15^{th} *ACM SOSP*, pp. 267–284, 1995.

[44] M. Fiuczynski and B. Bershad, "An Extensible Protocol Architecture for Application-Specific Networking," in *Proceedings of the 1996 Winter USENIX Conference*, Jan. 1996.

[45] 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.

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

[47] H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," in *Proceedings of OOPSLA '95*, (Austin, Texas), ACM, October 1995.

[48] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.

[49] M. Zitterbart, B. Stiller, and A. Tantawy, "A Model for High-Performance Communication Subsystems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 507–519, May 1993.

[50] I. Object-Oriented Concepts, "ORBacus User Manual - Version 3.1.2." www.ooc.com/ob, 1999.

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

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

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

[54] 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.

[55] 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.

[56] T. Harrison, D. C. Schmidt, A. Gokhale, and G. Parulkar, "Operating System Support for High-Performance, Real-time CORBA," in *Proceedings of the* 5^{th} *International Workshop on Object-Orientation in Operating Systems*, IEEE, October 1996.

[57] 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.

[58] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications," in *Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '93)*, (Lancaster, U.K., New Hampshire), pp. 35–48, November 1993.

[59] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

[60] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[61] Sun Microsystems, *STREAMS Programming Guide*. Sun Microsystems, Inc., Mountain View, CA, August 1997. Revision A.

[62] OSI Special Interest Group, *Transport Provider Interface Specification*, December 1992.

[63] OSI Special Interest Group, *Network Provider Interface Specification*, December 1992.

[64] OSI Special Interest Group, *Data Link Provider Interface Specification*, December 1992.

A Synopsis of CORBA

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 [51]. Figure 21 illustrates the key components in the CORBA reference model [52] that collaborate to provide this degree of portability, interoperability, and transparency.[4] Each component in the

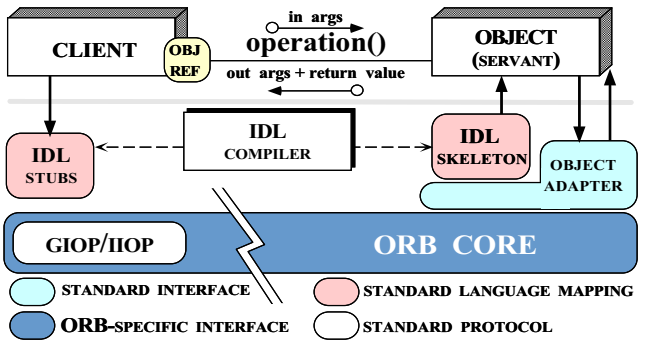

Figure 21: Key Components in the CORBA 2.x Reference Model

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 21 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

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 object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `structs`. A client never interacts with servants directly, but always through objects identified by object references.

[4] This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [52].

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), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [53] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern [53] and demarshal the message-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [54].

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

B Overview of Solaris

The Solaris kernel is a *preemptive, multi-threaded, real-time, and dynamically loaded* implementation of UNIX SVR4 and POSIX. It is designed to work on uni-processors and shared memory symmetric multi-processors. Solaris contains a real-time nucleus that supports multiple threads of control in the kernel. Most control flows in the kernel, including interrupts, are threaded [55]. Below, we summarize the Solaris scheduling model and communication I/O subsystem.

B.1 Synopsis of the Solaris Scheduling Model

The application-level programming model of Solaris supports multiple threads of control within a single application process. Solaris provides a two-level thread scheduling model that consists of an application-level scheduler in the threads library and a global system scheduler in the kernel. Application level threads are either bound to or scheduled to run on lightweight processes (LWPs), which can be thought of as virtual CPUs, by the threads library. In turn, each LWP is bound to one kernel thread (*kthreads*) which is scheduled by the global system scheduler to run on the available CPUs. Note, there are kernel threads which are not associated with any LWP, these are termed system threads.

The traditional UNIX scheduling policy targets time-sharing, interactive environments. This traditional scheduler is preemptive, time-sliced, priority based where the highest priority runnable thread is always scheduled. The priorities vary as a function of the threads CPU usage pattern: the more CPU time used by a thread the lower its priority. Thus, compute bound threads will have progressively lower priorities until some lower limit is reached. For threads with the same priority time slicing is used. Generally, the lower a threads priority the larger its time slice. Newer implementations of the time sharing class have additional parameters however the policy remains the same. While this approach is well suited for traditional time sharing UNIX environments it does not satisfy the scheduling needs for the new class of multimedia and real-time DOC applications [56, 57, 58].

Scheduling classes: Solaris extends the traditional UNIX time-sharing scheduler [16] to provide a flexible framework that allows dynamic linking of custom *scheduling classes*. For instance, it is possible to implement a new scheduling policy as a scheduling class and load it into a running Solaris kernel. By default, Solaris supports the four scheduling classes shown ordered by decreasing global scheduling priority below:

Scheduling Class	Priorities	Typical purpose
Interrupt (INTR)	160-169	Interrupt Servicing
Real-Time (RT)	100 - 159	Fixed priority scheduling
System (SYS)	60-99	OS-specific threads
Time-Shared (TS)	0-59	Time-Shared scheduling

The Time-Sharing (TS)[5] class is similar to the traditional UNIX scheduler [16], with enhancements to support interactive windowing systems. The System class (SYS) is used to schedule system kthreads, including I/O processing, and is not available to application threads. The Real-Time (RT) scheduling class uses fixed priorities above the SYS class. Finally, the

[5] In this discussion we include the Interactive (IA) class, which is used primarily by Solaris windowing systems, with the TS class because they share the same range of global scheduling priorities.

highest system priorities are assigned to the Interrupt (INTR) scheduling class [55].

By combining a threaded, preemptive kernel with a fixed priority real-time scheduling class, Solaris attempts to provide a worst-case bound on the time required to dispatch application threads or kernel threads [13]. The RT scheduling class supports both Round-Robin and FIFO scheduling of threads. For Round-Robin scheduling, a time quantum specifies the maximum time a thread can run before it is preempted by another RT thread with the same priority. For FIFO scheduling, the highest priority thread can run for as long as it chooses, until it voluntarily yields control or is preempted by an RT thread with a higher priority.

Timer mechanisms: Many kernel components use the Solaris timeout facilities. To minimize priority inversion, Solaris separates its real-time and non-real-time timeout mechanisms [13]. This decoupling is implemented via two callout queue timer mechanisms: (1) `realtime_timeout`, which supports real-time callouts and (2) `timeout`, which supports non-real-time callouts.

The real-time callout queue is serviced at the lowest interrupt level, after the current clock tick is processed. In contrast, the non-real-time callout queue is serviced by a thread running with a SYS thread priority of 60. Therefore, non-real-time timeout functions cannot preempt threads running in the RT scheduling class.

B.2 Synopsis of the Solaris Communication I/O Subsystem

The Solaris communication I/O subsystem is an enhanced version of the SVR4 STREAMS framework [46] with protocols like TCP/IP implemented using STREAMS modules and drivers. STREAMS provides a bi-directional path between application threads and kernel-resident drivers. In Solaris, the STREAMS framework has been extended to support multiple threads of control within a STREAM [59].

Below, we outline the key components of the STREAMS framework and describe how they affect communication I/O performance and real-time determinism.

General structure of a STREAM: A STREAM is composed of a STREAM head, a driver and zero or more modules linked together by read queues (RQ) and write queues (WQ), as shown in Figure 22. The STREAM head provides an interface between an application process and a specific instance of a STREAM in the kernel. It copies data across the user/kernel boundary, notifies application threads when data is available, and manages the configuration of modules into a STREAM.

Each module and driver must define a set of entry points that handle `open`/`close` operations and process STREAM messages. The message processing entry points are `put` and `svc`,

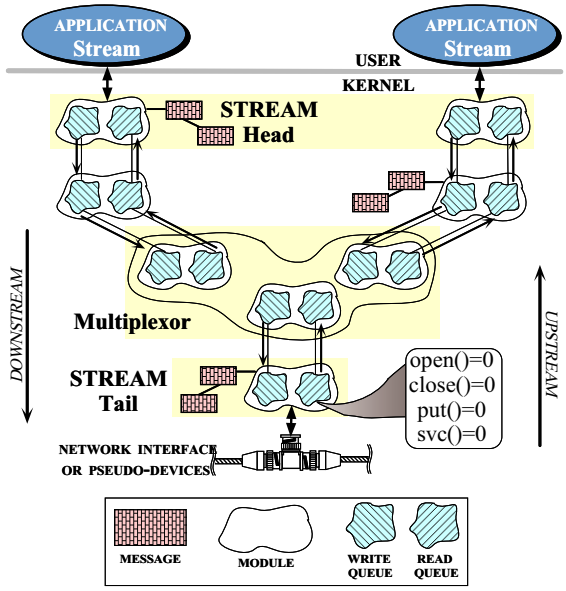

Figure 22: General Structure of a STREAM

which are referenced through the read and write queues. The `put` function provides the mechanism to send messages *synchronously* between modules, drivers, and the STREAM head.

In contrast, the `svc` function processes messages *asynchronously* within a module or driver. A background thread in the kernel's SYS scheduling class runs `svc` functions at priority 60. In addition, `svc` functions will run after certain STREAMS-related system calls, such as `read`, `write`, and `ioctl`. When this occurs, the `svc` function runs in the context of the thread invoking the system call.

Flow control: Each module can specify a high and low watermark for its queues. If the number of enqueued messages exceeds the HIGH_WATERMARK the STREAM enters the flow-controlled state. At this point, messages will be queued upstream or downstream until flow control abates.

For example, assume a STREAM driver has queued HIGH_WATERMARK+1 messages on its write queue. The first module atop the driver that detects this will buffer messages on its write queue, rather than pass them downstream. Because the STREAM is flow-controlled, the `svc` function for the module will not run. When the number of messages on the driver's write queue drops below the LOW_WATERMARK the STREAM will be re-enable automatically. At this point, the `svc` function for this queue will be scheduled to run.

STREAM **Multiplexors:** Multiple STREAMS can be linked together using a special type of driver called a *multiplexor*. A multiplexor acts like a driver to modules above it and as a STREAM head to modules below it. Multiplexors enable the STREAMS framework to support layered network protocol

stacks [60].

Figure 23 shows how TCP/IP is implemented using the Solaris STREAMS framework. IP behaves as a multiplexor by

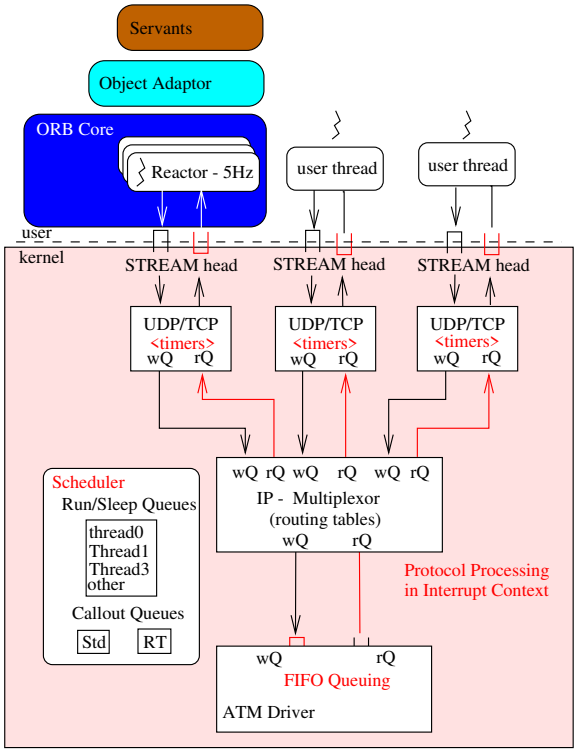

Figure 23: Conventional Protocol Stacks in Solaris STREAMS

joining different transport protocols with one or more link layer interfaces. Thus, IP demultiplexes both incoming and outgoing datagrams.

Each outgoing IP datagram is demultiplexed by locating its destination address in the IP routing table, which determines the network interface it must be forwarded to. Likewise, each incoming IP datagram is demultiplexed by examining the transport layer header in a STREAMS message to locate the transport protocol and port number that designates the correct upstream queue.

Multi-threaded STREAMS: Solaris STREAMS allows multiple kernel threads to be active in STREAMS I/O modules, drivers, and multiplexors concurrently [61]. This multi-threaded STREAMS framework supports several levels of concurrency, which are implemented using the *perimeters* [59] shown below:

Per-module with single threading
Per-queue-pair single threading
Per-queue single threading
Any of the above with unrestricted `put` and `svc`
Unrestricted concurrency

In Solaris, the concurrency level of IP is "per-module" with concurrent `put`, TCP and `sockmod` are "per-queue-pair," and UDP is "per-queue-pair" with concurrent `put`. These perimeters provide sufficient concurrency for common use-cases. However, there are cases where IP must raise its locking level when manipulating global tables, such as the IP routing table. When this occurs, messages entering the IP multiplexor are placed on a special queue and processed asynchronously when the locking level is lowered [59, 55].

Callout queue callbacks: The Solaris STREAMS framework provides functions to set timeouts and register callbacks. The `qtimeout` function adds entries to the standard non-real-time callout queue. This queue is serviced by a system thread with a SYS priority of 60, as described in Section B.1. Solaris TCP and IP use this callout facility for their protocol-specific timeouts, such as TCP keepalive and IP fragmentation/reassembly.

Another mechanism for registering a callback function is `bufcall`. The `bufcall` function registers a callback function that is invoked when a specified size of buffer space becomes available. For instance, when buffers are unavailable, `bufcall` is used by a STREAM queue to register a function, such as `allocb`, which is called back when space is available again. These callbacks are handled by a system thread with priority SYS 60.

Network I/O: The Solaris network I/O subsystem provides service interfaces that reflect the OSI reference model [60]. These service interfaces consist of a collection of primitives and a set of rules that describe the state transitions.

Figure 23 shows how TCP/IP is structured in the Solaris STREAMS framework. In this figure, UDP and TCP implement the Transport Protocol Interface (TPI) [62], IP the Network Provider Interface (NPI) [63] and ATM driver the Data Link Provider Interface (DLPI) [64]. Service primitives are used (1) to communicate control (state) information and (2) to pass data messages between modules, the driver, and the STREAM head.

Data messages (as opposed to control messages) in the Solaris network I/O subsystem typically follow the traditional BSD model. When an application thread sends data it is copied into kernel buffers, which are passed through the STREAM head to the first module. In most cases, these messages are then passed through each layer and into the driver through a nested chain of `puts` [59]. Thus, the data are sent to the network interface driver within the context of the sending process and typically are not processed asynchronously by module `svc` functions. At the driver, the data are either sent out immediately or are queued for later transmission if the interface is busy.

When data arrive at the network interface, an interrupt is generated and the data (usually referred to as a frame or

packet) is copied into kernel buffer. This buffer is then passed up through IP and the transport layer in interrupt context, where it is either queued or passed to the STREAM head via the socket module. In general, the use of `svc` functions is reserved for control messages or connection establishment.