The Performance of a Real-time I/O Subsystem for QoS-enabled ORB Middleware

Fred Kuhns, Douglas C. Schmidt, and David L. Levine
{fredk,schmidt,levine}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA [*]

Submitted to the International Symposium on Distributed Objects and Applications (DOA'99), Edinburgh, Scotland, 5th–6th September 1999.

Abstract

There is increasing demand to extend Object Request Broker (ORB) middleware to support applications with stringent quality of service (QoS) requirements. However, conventional ORBs do not define standard features for specifying or enforcing end-to-end QoS for applications with deterministic real-time requirements. This paper describes the design and performance of a real-time I/O (RIO) subsystem optimized for QoS-enabled ORB endsystems that support high-performance and real-time applications running on off-the-shelf hardware and software. The paper illustrates how integrating a real-time ORB with a real-time I/O subsystem can reduce latency bounds on end-to-end communication between high-priority clients without unduly penalizing low-priority and best-effort clients.

Keywords: Real-time CORBA Object Request Broker, Quality of Service for OO Middleware, Real-time I/O Subsystems.

1 Introduction

Object Request Broker (ORB) middleware like CORBA [1] and DCOM [2] is well-suited for request/response applications with best-effort quality of service (QoS) requirements. However, ORB middleware has historically been unsuited for performance-sensitive, distributed real-time applications. In general, conventional ORBs suffer from (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) lack of performance optimizations [3].

To address these shortcomings, we have developed *The ACE ORB* (TAO) [4]. TAO is a high-performance, real-time 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, communication protocol, and CORBA-compliant middleware components and features shown in Figure 1.

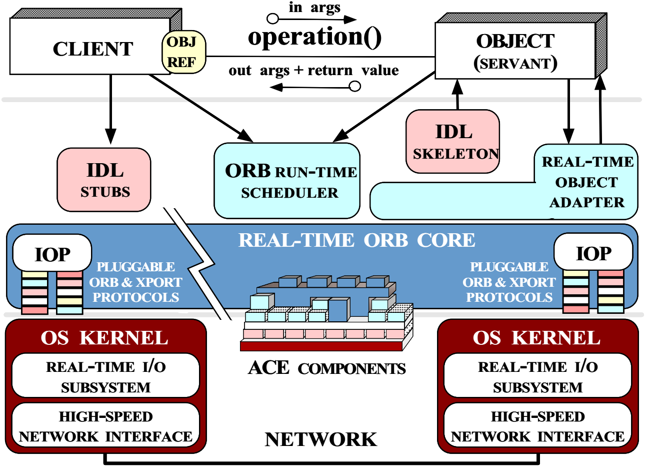


Figure 1: Components in the TAO Real-time ORB Endsystem

TAO's real-time I/O (RIO) subsystem runs in the OS kernel. It uses a pool of real-time threads to send/receive requests to/from clients across high-speed networks or I/O backplanes. TAO's ORB Core, Object Adapter, and servants run in user-space. TAO's ORB Core contains a pool of real-time threads that are co-scheduled with the RIO subsystem's thread pool. Together, these threads process client requests in accordance with their QoS requirements. TAO's Object Adapter uses perfect hashing [5] and active demultiplexing [6] to demultiplex these requests to application-level servant operations in constant, $O(1)$ time.

We have used TAO to research key dimensions of high-performance and real-time ORB endsystem design including static [4] and dynamic [7] scheduling, request demultiplexing [6], event processing [8], ORB Core connection and con-

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

currency architectures [9], IDL compiler stub/skeleton optimizations [10], and ORB performance [11]. This paper focuses on an essential, and previously unexamined, dimension in the real-time ORB endsystem design space: *the development and analysis of a real-time I/O (RIO) subsystem that supports QoS requirements for real-time CORBA applications.* This paper extends results in [12] that focus solely on I/O subsystem performance to illustrate empirically how the TAO real-time ORB endsystem benefits from our RIO enhancements to the Solaris 2.5.1 OS kernel.

The paper is organized as follows: Section 2.2 describes how the RIO subsystem enhances the Solaris 2.5.1 OS kernel to support end-to-end QoS for TAO applications; Section 3 presents empirical results from systematically benchmarking the efficiency and predictability of TAO and RIO over an ATM network; Section 4 compares RIO with related work; and Section 5 presents concluding remarks. For completeness, Appendix A presents a synopsis of the Solaris real-time scheduling model and communication I/O subsystem.

2 The Design of TAO's Real-time I/O Subsystem on Solaris over ATM

In this section, we examine the components that affect the performance and determinism of the RIO subsystem. The main focus of our research is on alleviating key sources of ORB endsystem priority inversion to increase application and middleware determinism.

2.1 Pros and Cons of Solaris for Real-time ORB Middleware

Below, we outline the pros and cons of using Solaris for real-time ORB middleware.

2.1.1 Motivation for Using Solaris

TAO's original real-time I/O subsystem ran over a proprietary VME backplane protocol integrated into VxWorks running on a 200 MHz PowerPC CPU [8]. All protocol processing was performed at interrupt-level in a VxWorks device driver. This design was optimized for low latency, *e.g.*, one-way ORB operations were $\sim$300 μsecs end-to-end, with $\sim$100 μsecs spent in the ORB and μsecs spent in the OS and VME backplane [13].

Unfortunately, the VME backplane driver is not portable to a broad range of real-time systems. Moreover, it is not suitable for more complex transport protocols, such as TCP/IP, which cannot be processed entirely at interrupt-level without incurring excessive priority inversion [14]. Therefore, we developed a real-time I/O (RIO) subsystem that is integrated into a standard protocol processing framework: the STREAMS [15] communication I/O subsystem on Solaris.

We used Solaris as the basis for our research on TAO and RIO for the following reasons:

Real-time support: Solaris attempts to minimize dispatch latency [16] for real-time threads. Moreover, its fine-grained locking of kernel data structures allows bounded thread preemption overhead.

Multi-threading support: Solaris supports a scalable multi-processor architecture, with both kernel-level (kthreads) and user-level threads.

Dynamic configurability: Most Solaris kernel components are dynamically loadable modules, which simplifies debugging and testing of new kernel modules and protocol implementations.

Compliant to open system standards: Solaris supports the POSIX 1003.1c [17] real-time programming application programming interfaces (APIs), as well as other standard POSIX APIs for multi-threading [18] and communications.

Tool support: There are many software tools and libraries [19] available to develop multi-threaded distributed, real-time applications on Solaris.

Availability: Solaris is widely used in research and industry.

Kernel source: Sun licenses the source code to Solaris, which allowed us to modify the kernel to support the multi-threaded, real-time I/O scheduling class described in Section 2.2.

In addition to Solaris, TAO runs on a wide variety of real-time operating systems, such as LynxOS, VxWorks, and Sun/Chorus ClassiX, as well as general-purpose operating systems with real-time extensions, such as Digital UNIX, Windows NT, and Linux. We plan to integrate the RIO subsystem architecture described in this section into other operating systems that implement the STREAMS I/O subsystem architecture.

2.1.2 Limitations of Solaris for Real-time ORB Middleware

Below, we review the limitations of Solaris when it is used as the I/O subsystem for real-time ORB endsystems.[1] These limitations stem largely from the fact that the Solaris RT scheduling class is not well integrated with the Solaris STREAMS-based network I/O subsystem. In particular, Solaris only supports the RT scheduling class for CPU-bound user threads, which yields the priority inversion hazards for real-time ORB endsystems described in Sections 2.1.2 and 2.1.2.

[1] Appendix A outlines the structure and functionality of the existing Solaris 2.5.1 scheduling model and communication I/O subsystem.

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 [20]. In Solaris, this type of priority inversion generally occurs when real-time user threads depend on kernel processing that is performed at the SYS or INTR priority levels [16, 14, 20]. Priority inversion may not be a general problem for user applications with "best-effort" QoS requirements. It is problematic, however, for real-time applications that require bounded processing times and strict deadline guarantees.

The Solaris STREAMS framework is fraught with opportunities for thread-based priority inversion, as described below:

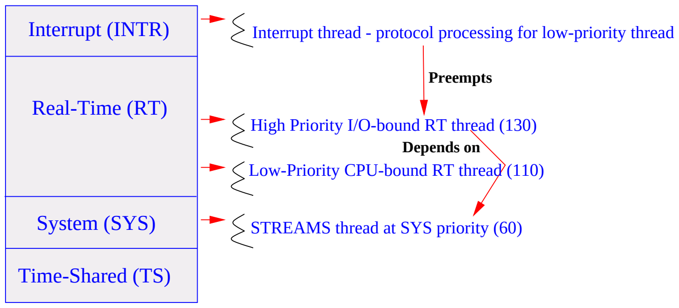
• **STREAMs-related `svc` threads:** When used inappropriately, STREAMS `svc` functions can yield substantial unbounded priority inversion. The reason is that `svc` functions are called from a kernel `svc` thread, known as the STREAMS background thread. This thread runs in the SYS scheduling class with a global priority of 60.

In contrast, real-time threads have priorities ranging from 100 to 159. Thus, it is possible that a CPU-bound RT thread can starve the `svc` thread by monopolizing the CPU. In this case, the `svc` functions for the TCP/IP modules and multiplexors will not run, which can cause unbounded priority inversion.

For example, consider a real-time process control application that reads data from a sensor at a rate of 20 Hz and sends status messages to a remote monitoring system. Because this thread transmits time-critical data, it is assigned a real-time priority of 130 by TAO's run-time scheduler. When this thread attempts to send a message over a flow-controlled TCP connection, it will be queued in the `TCP` module for subsequent processing by the `svc` function.

Now, assume there is another real-time thread that runs asynchronously for an indeterminate amount of time responding to external network management trap events. This asynchronous thread has an RT priority of 110 and is currently executing. In this case, the asynchronous RT thread will prevent the `svc` function from running. Therefore, the high-priority message from the periodic thread will not be processed until the asynchronous thread completes, which can cause the unbounded priority inversion depicted in Figure 2.

In addition, two other STREAMS-related system kthreads can yield priority inversions when used with real-time applications. These threads run with a SYS priority of 60 and handle the callback functions associated with the `bufcall` and `qtimeout` system functions described in Section A.2. This problem is further exacerbated by the fact that the priority of the thread that initially made the buffer request is not considered when these `svc` threads process the requests on their respective queues. Therefore, it is possible that a lower priority connection can receive buffer space before a higher priority



Interrupt (INTR)
Real-Time (RT)
System (SYS)
Time-Shared (TS)
Interrupt thread - protocol processing for low-priority thread
Preempts
High Priority I/O-bound RT thread (130)
Depends on
Low-Priority CPU-bound RT thread (110)
STREAMS thread at SYS priority (60)

Figure 2: Common Sources of Priority Inversion in Solaris

connection.

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

In BSD UNIX-based systems [21], 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 user 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 systems, protocol processing can either be performed at 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 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 that if protocol processing on incoming data is performed in interrupt context this can lead to a condition known as *receive* [14]. Livelock is a condition where the overall endsystem performance degrades due to input processing of packets at 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.

Packet-based priority inversions: Packet-based priority inversion can occur when packets for high-priority applications are queued behind packets for low-priority user threads. In the Solaris I/O subsystem, for instance, this 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.

Although TCP/IP in Solaris is multi-threaded, it incurs packet-based priority inversion because it enqueues network data in FIFO order. For example, TAO's priority-based ORB Core, described in [12], associates all packets destined for a particular TCP connection with a real-time thread of the appropriate priority. However, different TCP connections can be associated with different thread priorities. Therefore, packet-based priority inversion will result when the OS kernel places packets from different connections in the same queue and processes sequentially. Figure 10 depicts this case, where the queues shared by all connections reside in the IP multiplexor and interface driver.

To illustrate this problem, consider an embedded system where Solaris is used for data collection and fault management. This system must transmit both (1) high-priority real-time network management status messages and (2) low-priority bulk data radar telemetry. For the system to operate correctly, status messages must be delivered periodically with strict bounds on latency and jitter. Conversely, the bulk data transfers occur periodically and inject a large number of radar telemetry packets into the I/O subsystem, which are queued at the network interface.

In Solaris, the packets containing high-priority status messages can be queued in the network interface *behind* the lower priority bulk data radar telemetry packets. This situation yields packet-based priority inversion. Thus, status messages may arrive too late to meet end-to-end application QoS requirements.

2.2 RIO – An Integrated I/O Subsystem for Real-time ORB Endsystems

Meeting the requirements of distributed real-time applications requires more than defining QoS interfaces with CORBA IDL or developing an ORB with real-time thread priorities. Instead, it requires the integration of the ORB and the I/O subsystem to provide end-to-end real-time scheduling and real-time communication to CORBA applications. This section describes how we have developed a real-time I/O (RIO) subsystem for TAO that alleviates the limitations of Solaris described in Section 2.1 by customizing the Solaris 2.5.1 OS kernel to support real-time network I/O over ATM/IP networks [22].

Enhancing a general-purpose OS like Solaris to support the QoS requirements of a real-time ORB endsystem like TAO requires the resolution of the following design challenges:

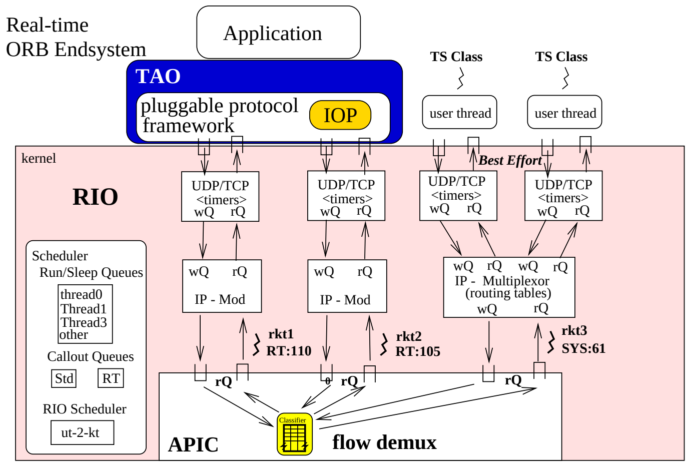


Figure 3: Architecture of the RIO Subsystem and Its Relationship to TAO

1. Creating an extensible and predictable I/O subsystem framework that can integrate seamlessly with a real-time ORB.
2. Alleviating key sources of packet-based and thread-based priority inversion.
3. Implementing an efficient and scalable packet classifier that performs early demultiplexing in the ATM driver.
4. Supporting high-bandwidth network interfaces, such as the APIC [23].
5. Supporting the specification and enforcement of QoS requirements, such as latency bounds and network bandwidth.
6. Providing all these enhancements to applications via the standard STREAMS network programming APIs [24].

This section describes the RIO subsystem enhancements we applied to the Solaris 2.5.1 kernel to resolve these design challenges. Our RIO subsystem enhancements provide a highly predictable OS run-time environment for TAO's integrated real-time ORB endsystem architecture, which is shown in Figure 3.

Our RIO subsystem enhances Solaris by providing QoS specification and enforcement features that complement TAO's priority-based concurrency and connection architecture [25]. The resulting real-time ORB endsystem contains user threads and kernel threads that can be scheduled statically. As described in [4], TAO's static scheduling service runs off-line to map periodic thread requirements and task dependencies to a set of real-time global thread priorities. These priorities are then used on-line by the Solaris kernel's run-time scheduler to dispatch user and kernel threads on the CPU(s).

To develop the RIO subsystem and integrate it with TAO, we extended our prior work on ATM-based I/O subsystems to provide the following features:

Early demultiplexing: This feature associates packets with the correct priorities and a specific STREAM early in the packet processing sequence, *i.e.*, in the ATM network interface driver [23]. 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.

Schedule-driven protocol processing: This feature performs all protocol processing in the context of kernel threads that are scheduled with the appropriate real-time priorities [26, 27, 28, 14]. RIO's design schedules network interface bandwidth and CPU time to minimize priority inversion and decrease interrupt overhead during protocol processing.

Dedicated STREAMS**:** This feature isolates request packets belonging to different priority groups to minimize FIFO queueing and shared resource locking overhead [29]. RIO's design resolves resource conflicts that can otherwise cause thread-based and packet-based priority inversions.

A complete description of RIO's RIO's components and features appears in [12]. Section 3.4 summarizes how the features in RIO alleviate the limitations with the original Solaris' I/O subsystem outlined in Section 2.1.

3 Empirical Benchmarking Results

Our earlier work [12] 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 [9].

3.1 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 interface to an OC3 155 Mbps port on the ASX-1000. The testbed configuration is shown in Figure 4.

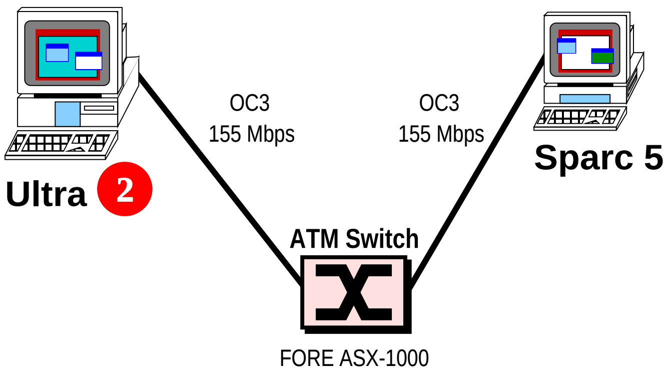


Figure 4: End-to-End ORB Endsystem Benchmark

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

3.2.1 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 4, 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 4, 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 5. With the RIO subsystem, high- and low-priority requests are

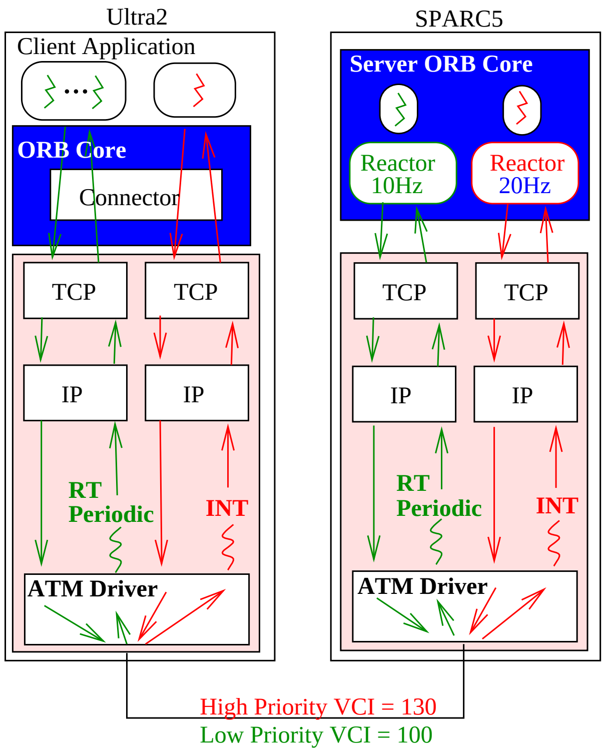


Figure 5: ORB Endsystem Benchmarking Configuration

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.

3.3 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 6 illustrates the average latency results for the high- and low-priority clients both with and without RIO. This figure

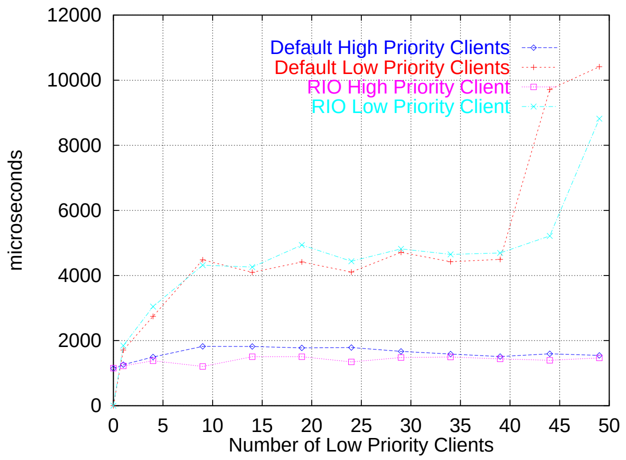


Figure 6: Measured Latency for All Clients with and without RIO

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 7 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 illustrates 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, ser-

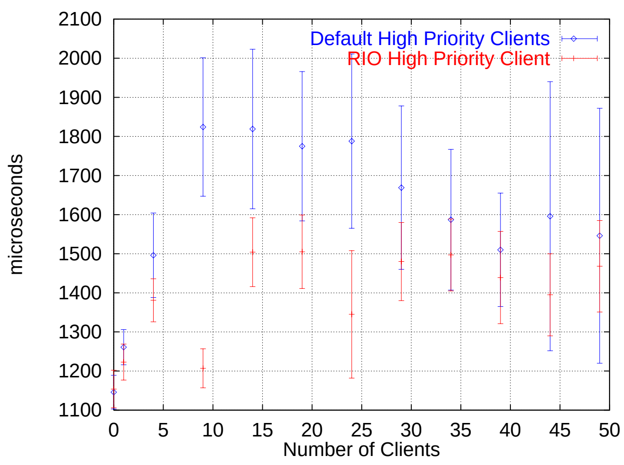


Figure 7: High-priority Client Latency and Jitter

vant 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 8 shows the average latency of low-priority client threads. This figure illustrates that the low-priority clients in-

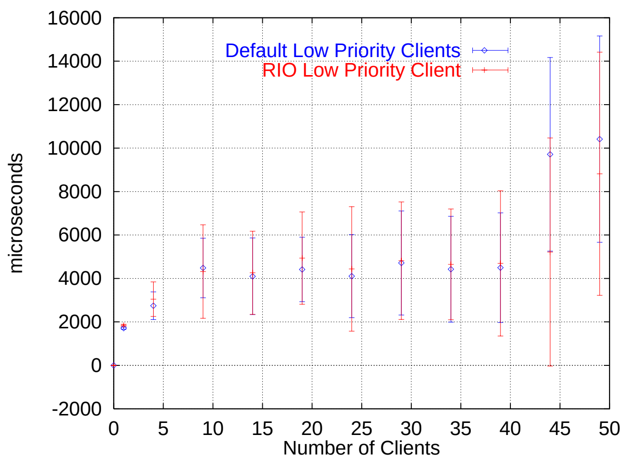


Figure 8: Low-priority Client Latency

curred 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.

3.4 Summary of Empirical Results

Our empirical results presented in Section 3 illustrate how RIO provides the following benefits to real-time ORB endsystems:

1. Reduced latency and jitter: RIO reduces round-trip latency and jitter for real-time network I/O, even during high network utilization. RIO prioritizes network protocol processing to ensure resources are available when needed by real-time applications.

2. Enforced bandwidth guarantees: The RIO periodic processing model provides network bandwidth guarantees. RIO's schedule-driven protocol processing enables an application to specify periodic I/O processing requirements which are used to guarantee network bandwidth.

3. Fine-grained resource control: RIO enables fine-grained control of resource usage, *e.g.*, applications can set the maximum throughput allowed on a per-connection basis. Likewise, applications can specify their priority and processing requirements on a per-connection basis. TAO also uses these specifications to create off-line schedules for statically configured real-time applications.

4. End-to-end priority preservation: RIO preserves end-to-end operation priorities by co-scheduling TAO's ORB Reactor threads with RIO kthreads that perform I/O processing.

5. Supports best-effort traffic: RIO supports the four QoS features described above without unduly penalizing best-effort, *i.e.*, traditional network traffic. RIO does not monopolize the system resources used by real-time applications. Moreover, because RIO does not use a fixed allocation scheme, resources are available for use by best-effort applications when they are not in use by real-time applications.

4 Related Work on I/O Subsystems

Our real-time I/O (RIO) subsystem incorporates advanced techniques [30, 23, 28, 29, 31] for high-performance and real-time protocol implementations. This section compares RIO with related work on I/O subsystems.

I/O subsystem support for QoS: The Scout OS [32, 33] 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 [26]. 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 [8] with deterministic QoS requirements.

SPIN [34, 35] 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 typesafe 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.

[29] 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 interrupt context is not addressed.

[14, 28] 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 [28], 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.

5 Concluding Remarks

This paper focuses on 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 of the TAO ORB endsystem. RIO provides QoS guarantees for vertically integrated ORB endsystems in order to increase (1) throughput and latency performance and (2) end-to-end determinism. RIO supports periodic protocol processing, guarantees I/O resources to applications, and minimizes the effect of flow control in a STREAM.

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

We learned the following lessons from our integration of RIO and TAO:

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. 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 [9] and Object Adapter [36], which are designed explicitly to preserve end-to-end QoS properties in distributed real-time systems [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 [37] 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 ORB to support end-to-end QoS guarantees over ATM/IP networks [22].

The TAO and RIO integration focused initially on statically scheduled applications with deterministic QoS requirements. we have subsequently extended the TAO ORB endsystem to support dynamic scheduling [7] and applications with statistical QoS requirements. The C++ source code for ACE, TAO, and our benchmarks is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`. The RIO subsystem is available to Solaris source licensees.

The TAO research effort has influenced the OMG Realtime CORBA specification [38], which was recently adopted as a CORBA standard. We continue to track the process of this standard and to contribute its evolution.

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] 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.

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

[5] D. C. Schmidt, “GPERF: A Perfect Hash Function Generator,” in *Proceedings of the 2^{nd} C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

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

[7] D. L. L. Christopher D. Gill 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*, 1999, to appear.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] C. O’Ryan, F. Kuhns, D. C. Schmidt, and J. Parsons, “Applying Patterns to Design a High-performance, Real-time Pluggable Protocols Framework for OO Communication Middleware,” in *Submitted to the OOPSLA ’99*, (Denver, CO), ACM, Oct. 1999.

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

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

[17] “Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language],” 1995.

[18] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[19] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems,” *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[20] 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.

[21] 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.

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

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

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

[25] 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*, To appear 1999.

[26] C. Cranor and G. Parulkar, “Design of Universal Continuous Media I/O,” in *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV ’95)*, (Durham, New Hampshire), pp. 83–86, Apr. 1995.

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

[28] 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.

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

[30] 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.

[31] J. Mogul and S. Deering, “Path MTU Discovery,” *Network Information Center RFC 1191*, pp. 1–19, Apr. 1990.

[32] A. B. Montz, D. Mosberger, S. W. O’Malley, L. L. Peterson, T. A. P. sting, and J. H. Hartman, “Scout: A communications-oriented operating system,” Tech. Rep. 94-20, Department of Computer Science, University of Arizona, June 1994.

[33] D. Mosberger and L. Peterson, “Making Paths Explicit in the Scout Operating System,” in *Proceedings of OSDI ’96*, Oct. 1996.

[34] B. Bershad, “Extensibility, Safety, and Performance in the Spin Operating System,” in *Proceedings of the* 15th *ACM SOSP*, pp. 267–284, 1995.

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

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

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

[38] Object Management Group, *Realtime CORBA 1.0 Joint Submission*, OMG Document orbos/98-12-05 ed., December 1998.

[39] 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.

[40] 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.

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

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

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

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

A 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 [39]. Below, we summarize the Solaris scheduling model and communication I/O subsystem.

A.1 Synopsis of the Solaris Scheduling Model

Scheduling classes: Solaris extends the traditional UNIX time-sharing scheduler [21] 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)[2] class is similar to the traditional UNIX scheduler [21], 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 user threads. The Real-Time (RT) scheduling class uses fixed priorities above the SYS class. Finally, the highest system priorities are assigned to the Interrupt (INTR) scheduling class [39].

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 user threads or kernel threads [16]. 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 [16]. 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.

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

The Solaris communication I/O subsystem is an enhanced version of the SVR4 STREAMS framework [15] with protocols like TCP/IP implemented using STREAMS modules and drivers. STREAMS provides a bi-directional path between user threads and kernel-resident drivers. In Solaris, the STREAMS framework has been extended to support multiple threads of control within a STREAM [40].

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

[2]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.

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 9. The STREAM head provides an interface

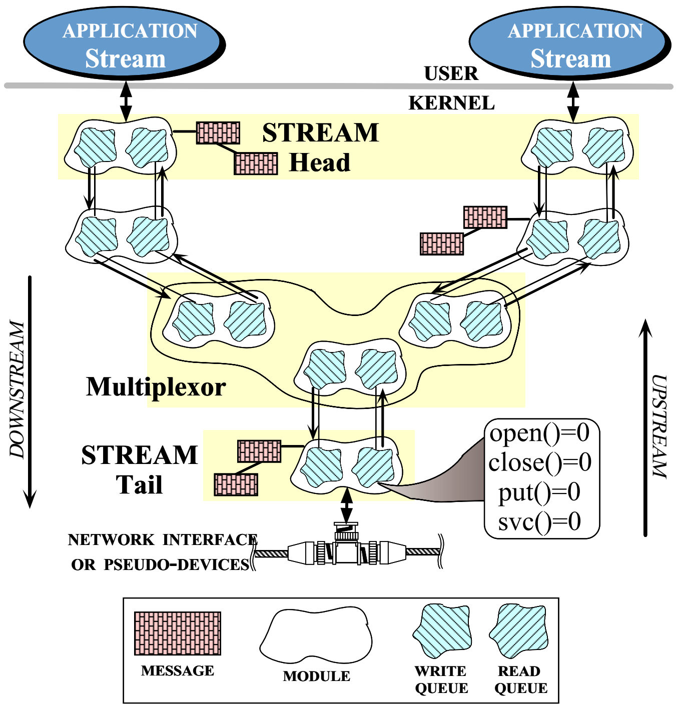


Figure 9: General Structure of a STREAM

between a user process and a specific instance of a STREAM in the kernel. It copies data across the user/kernel boundary, notifies user 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`, 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 [24].

Figure 10 shows how TCP/IP is implemented using the Solaris STREAMS framework. IP behaves as a multiplexor by

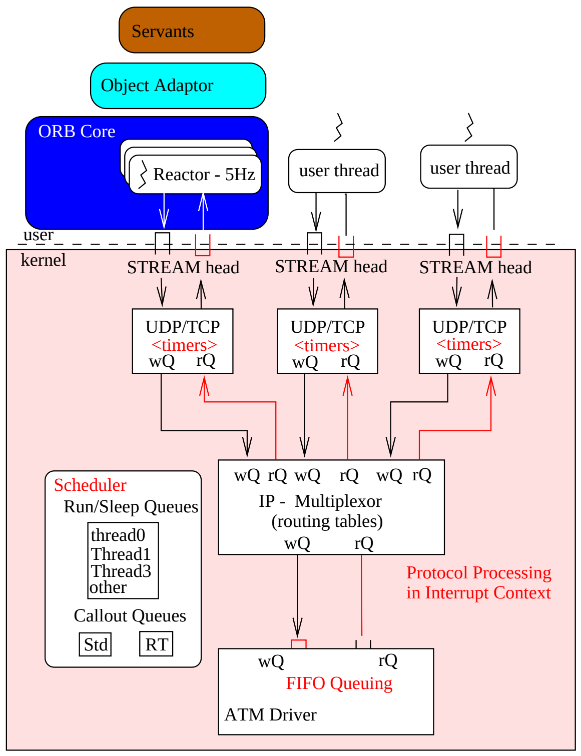


Figure 10: Conventional Protocol Stacks in Solaris STREAMS

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 [41]. This multi-threaded STREAMS framework supports several levels of concurrency, which are implemented using the *perimeters* [40] 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 [40, 39].

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 A.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 [24]. These service interfaces consist of a collection of primitives and a set of rules that describe the state transitions.

Figure 10 shows how TCP/IP is structured in the Solaris STREAMS framework. In this figure, UDP and TCP implement the Transport Protocol Interface (TPI) [42], IP the Network Provider Interface (NPI) [43] and ATM driver the Data Link Provider Interface (DLPI) [44]. 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 a user 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` [40]. 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.